PROGRAMMING AND CODING FOR AUTOMATIC DIGITAL COMPUTERS

McGraw-Hill Series in Information Processing and Computers

J. P. NASH, CONSULTING EDITOR

DAVIS · Computability and Unsolvability

EVANS AND PERRY · Programming and Coding for Automatic Digital Computers

GOTLIEB AND HUME · High-speed Data Processing

KARPLUS · Analog Simulation

MCCORMICK · Digital Computer Primer

ROGERS AND CONNOLLY · Analog Computation in Engineering Design

SCOTT · Analog and Digital Computer Technology

WILLIAMS · Digital Computing Systems

WRUBEL · A Primer of Programming for Digital Computers

PROGRAMMING AND CODING FOR AUTOMATIC DIGITAL COMPUTERS

G. W. EVANS II

Manager, Mathematical Sciences Department
Stanford Research Institute

C. L. PERRY

Consultant, Mathematical Sciences Department, Stanford Research Institute
Director of the Computation Facility, University of California, San Diego

CHAPTER 8: SOME ASPECTS OF AUTOMATIC PROGRAMMING

By R. E. Keirstead, Jr.

Head, Computation Group, Mathematical Sciences Department
Stanford Research Institute

McGRAW-HILL BOOK COMPANY, INC.

New York Toronto London

1961

PROGRAMMING AND CODING FOR AUTOMATIC DIGITAL COMPUTERS

 Library of Congress Catalog Card No.: 60-53353

19755

THE MAPLE PRESS COMPANY, YORK, PA.

To

John von Neumann

PREFACE

The preparation of problems for digital computers can be divided into phases such as: problem formulation, selection of numerical methods, programming and coding, and operation of the computer. The need for books presenting various systematic approaches to these interrelated phases becomes greater each year with the rapidly expanding use of digital computers as a research tool (evaluating solutions for particular problems in mathematics, physical and biological sciences, engineering and other fields) and as a production tool (in accounting, inventory, and the development and testing of equipment designs). Enumeration and description of applications in which digital computers are used and may be used is the subject matter for many books of fact and fiction as well as journal articles and short stories. This book, however, is devoted primarily to descriptions of some of the methods used in the programming and coding phase of computer use. The programming and coding phase deals with the development of instructions for the steps the computer is to execute automatically in performing the computation desired. Only minor consideration is given to problems in the other phases of use of computers, although the proper use of digital computers requires that appropriate attention be given the important phases of problem formulation, selection of numerical methods, and operation of the computer. The methods proposed in this book are based on use of the flow chart, as proposed by John von Neumann, to describe the sequence of the computation. It is interesting to note that another recently published book, "Mathematical Methods for Digital Computers" by Ralston and Wilf, also uses the flow chart as a primary link in describing a computational process. If this book satisfies a small part of the need for organized approaches to basic programming and coding, the authors will feel rewarded for their efforts.

Basic programming and coding is for the use of the computer in terms of the operations it was designed to execute. Automatic programming and automatic coding are terms describing methods of using the computer by utilizing sets of computer operations. Through basic programming

and coding procedures, efficient operation of the computer can be attained. Automatic programming and coding techniques frequently effect significant savings in programming effort at the expense of some computer efficiency. An understanding of the basic operation of the computer is essential for competent use, for only then can the user determine the details of faulty computer operation, design efficient computation codes, and construct automatic coding procedures.

Although considerable effort has been expended to make the use of digital computers easy by utilizing automatic coding procedures, these techniques are still in a development stage. Recently, the Association for Computing Machinery, under the direction of John Carr III, has participated in the formation of an international committee to determine some degree of standardization for the language of automatic computer procedures. At present almost all computation facilities use some automatic procedures to facilitate programming and coding, but these procedures vary from installation to installation. Frequently, users of equipment made by the same manufacturer cannot exchange computer programs because of differing internal computer operation, computer modifications, and input-output modifications. It is hoped that the efforts of John Carr III and others to standardize computing languages will result in at least the uniformity that now exists in mathematics, and that eventually considerable exchange of programs between computer installations will be possible. A first major result of this work is the publication of computer programs in the language ALGOL by the Association for Computing Machinery. ALGOL is becoming a base language for the communication of computer programs. Programs written in this language can then be translated into the systems for particular computers. Successful automatic programming systems, although currently incompatible, have been developed by Bendix, Burroughs, Control Data Corporation, IBM, Minneapolis Honeywell, Philco, Remington Rand, and others. The extensive use of automatic programming techniques is beginning to affect the internal design of computers. It is very likely that many of the next generation of computers will be designed to facilitate these techniques. Chapter 8 of this book discusses some aspects of automatic programming and coding techniques. Even though they might rapidly become obsolete, several books describing more of these procedures and their design would be well worth the writing at this time. The book-length Chapter 2 (270 pages) of the "Handbook of Automation, Computation and Control," edited by Grabbe, and the book "Programming Programme for the B.E.S.M. Computer" by Ershov are excellent beginnings for descriptions of the

field of automatic programming. The "Handbook of Automation, Computation and Control" also contains a glossary of computer terminology that may be helpful to beginning programmers.

It is assumed that the reader has had some college mathematics and intends to learn coding and programming in order to perform or administer useful work on automatic digital computers. We believe that the book will be useful to people associated with computer installations and that it is an appropriate junior-year college text for a course on coding and programming. Some chapters outline only general procedures, since a detailed discussion requires an intimate knowledge of the particular computer to be used; and an attempt has been made to avoid, except for demonstrating coding procedures, specific mention of a particular computer.

The internal cross references in the text have been organized on the following basis: Sections are double numbered by chapter. The equations, figures, and tables are triple numbered consecutively within sections of a chapter; the first number is the chapter number, the second the section number, and the third that of the equation, figure, or table; for example, Eq. (1-4-5) is the fifth equation of the fourth section of Chap. 1. The same system is used for text references to figures and tables.

The writing of this book reflects more than twelve years' experience for each author in the computer field and in allied fields of endeavor. Much of this experience was gained through contact, direct and indirect, with many persons who have furthered the development of computers and computer techniques. To acknowledge all who have contributed to the science of programming and coding (and, thus, the authors feel, to this book) would require a bibliographical search worthy of separate publication. The authors are not undertaking this task. However, not to mention a few contributors would be an act of ingratitude. It is hoped that those whose names we have omitted will understand that they are also acknowledged and will forgive our oversight.

Our first mention is of John von Neumann, H. H. Goldstine, and A. W. Burks whose Princeton reports[1] were the first integrated analysis of the coding, programming, and design requirements for an internally coded automatic digital computer. Others who, in our initial association with automatic computers, influenced our ways of thinking were D. A. Flanders, A. S. Householder, J. H. Alexander, J. C. Chu, and Mrs. Jean Hall. Certainly, we have been influenced by the works of M. V. Wikes, D. J. Wheeler, S. Gill, J. P. Eckert, J. W. Mauchly, H. H. Aiken, and many others who have been lost in our memories.

The authors would like to make special acknowledgment to T. H.

Morrin, General Manager, Engineering, Stanford Research Institute, whose continual encouragement made this book possible. We are further indebted to R. E. Keirstead, G. F. Wallace, and J. P. Nash for comments and suggestions concerning the manuscript; and to Mrs. Patricia Hamilton, Mrs. Elayne Carlson, and particularly Mrs. Jeanne DeWaal for the preparation of the manuscript. Finally, the authors are pleased to be able to include Chap. 8, "Some Aspects of Automatic Programming," which was written for this book by R. E. Keirstead of Stanford Research Institute.

G. W. Evans II
C. L. Perry

[1] John von Neumann, A. W. Burks, and H. H. Goldstine, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," part I, vol. I, Contract W-36-034-ORD-7481, U.S. Army Ord. Dept., June 28, 1946. John von Neumann and H. H. Goldstine, "Planning and Coding of Problems for an Electronic Computing Instrument," part II, vol. I, *ibid.*, 1947; part II, vols. II and III, *ibid.*, 1948.

These reports, which have been out of print for almost ten years, are available in some libraries. Together with other papers on automata, they are to be printed in one volume of "The Collected Works of John von Neumann," to be edited by A. H. Taub and published by Pergamon Press.

CONTENTS

1

SOME BASIC CONCEPTS IN PROGRAMMING AND CODING

1-1. Introduction

The modern automatic digital computers with which this book will be concerned have the ability to remember and to calculate. The ability to remember or preserve information consists of the storage and recall of digital information. Of the types of calculations, there are four main classes of operations. They are those operations involving additions, those involving multiplications, those involving divisions, and those involving comparisons. For the class of operations involving additions, the computer may have the ability to add a number to another number, to add the negative of a number to (subtract a number from) another number, to add the absolute value of a number to another number, and to add the negative of the absolute value (subtract the absolute value) of a number to another number. The computer may have the ability to produce a rounded or an unrounded product and a rounded or an unrounded quotient. Its comparison abilities are usually in the form of determining which is the larger of two designated numbers and, in accordance with this decision, choosing which of two alternative actions to follow. It may also distinguish whether or not two numbers are identical and, depending upon this result, make an appropriate choice of action to be followed. The computer may also perform certain logical operations which facilitate the coding of problems, e.g., the extraction of specified digits from a number and the substitution of these in another number. Often, computers use two types of number representations, fixed point and floating point, which will be discussed in detail in Chap. 2, and perform the aforementioned operations with either type. Furthermore, many computers handle digital codes for alphabetical as well as numerical information.

The tasks that may be assigned to a digital computer are limited by the computer's ability to calculate and by its memory capacity, i.e., its capacity to store and recall information. At first glance, its limited mathematical abilities appear to be a severe limitation. However, when we reflect that many of the more sophisticated mathematical operations may be approximated to any desired degree of accuracy by a sequence of arithmetic operations by the computer, we find the scope of problems which may be attacked to be quite large. For example, a definite integral may be defined in terms of a limit of a sum of products; a derivative may be defined in terms of a limit of a ratio, and a vector may be defined by its components. Even though the computer cannot perform a limiting process, it may, through competent guidance, approximate these limiting processes. Thus, through appropriate approximations, the computer will solve particular problems involving partial differential equations, systems of ordinary differential equations, integral equations, etc. Here the word "solve" is used in the practical sense, that is, approximation within a known error.

The basic computer operations are readily adapted for many problems involving stochastic (i.e., chance) processes and many of the basic statistical procedures may be appropriately defined in terms of sequences of the computer operations. Computers which handle alphabetical information as well as numerical information are used for payroll, inventory, billing, and other business applications.

The *programming* and *coding* of an automatic high-speed digital computer are the procedures used in the development of sequences of computer operations for the computer; they deal with the processing of information and data supplied to a computer in such a manner that it returns useful information. Rather than give rigorous definitions for the terms *programming* and *coding* at this time, we will just say that coding is one of the phases of programming. In general, a process for organizing and sequencing calculations for a high-speed computer is called programming.

During the past two decades developments in electronic and electromagnetic design have provided means of producing arithmetic calculators that have the ability to obtain the sum or difference of two numbers in a fraction of a microsecond and the product or quotient of two numbers in microseconds. The present-day electromechanical desk calculators can produce the sum or difference of two numbers in 0.1 second and the product or quotient of two numbers in an average time of 8 seconds. These speeds, however, have very little meaning until we state the size of the numbers involved.

For a foundation, let us review some basic definitions. The *decimal*

digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. A number may be represented in the decimal system by a sequence of one or more of these digits. For example, the number three thousand two hundred and ninety-nine is represented by the four decimal digits 3,299. We now introduce two terms which at first might be considered synonyms but which have very different meanings. The first of these is the *size* or *length* of a number representation. By *size* we will mean the number of digits appearing in the representation for the number. The second we will call the *magnitude* of a number which tells us how many units are contained in the number. The two numbers 43,270 and 30,472 contain not only the same number of digits, but also the same digits, and each representation is five digits long (size). However, 43,270 is larger in magnitude than 30,472 since it contains 12,798 more units.

In addition to its magnitude, a number may be assigned a *sign*, plus (+) or minus (−), which tells whether the number is greater than zero or less than zero. We wish to point out that we do not intend at this time to give rigorous definitions, but only descriptions, and that the rigorous definitions will follow when we speak of number bases. The decimal representations for numbers are said to be of base 10. Later we will be forced to consider other systems of representing numbers.

Let us again consider the speeds of performing the arithmetic operations. The speeds previously stated in the case of the desk calculator, as well as the high-speed arithmetic unit of an electronic computer, are those for forming the sum or difference of two 10-decimal-digit numbers; multiplying two 10-decimal-digit numbers together to produce, at most, a 20-decimal-digit product; and dividing a 20-decimal-digit dividend by a 10-decimal-digit divisor to produce, at most, a 10-decimal-digit quotient and a 10-decimal-digit remainder. Considering the 8-second division time of a desk calculator, we find this speed only slightly slower than the human speed for writing down ten digits, and, since the quotient appears serially, digit by digit, the operator can write down the quotient as it appears with little wasted time. Thus the human operator and the desk calculator are comparable machines timewise, and there is little reason to improve the speeds of this type of desk calculator.

If high-speed computers consisted only of an arithmetic unit and registers for displaying results, one would find that the machine is 260,000 times faster than the human (allowing 4 seconds for the human to write down a 10-decimal-digit answer and assuming a computer multiplication speed of 15×10^{-6} seconds). Thus, the human is too slow to be coupled to a high-speed arithmetic unit. Furthermore, in practice, a human with a desk calculator makes at least one error per day. For an

average of one operation every 4 seconds, this represents an error every 7,200 operations. The high-speed arithmetic unit can perform 7,200 multiplications in about 0.108 second. Even if the human operator could keep pace with the high-speed arithmetic unit and even if he made only one mistake for every 66,700 computer operations, this would produce an error every second, or 28,800 errors per 8-hour day. High-speed computers perform arithmetic operations for days, some even for weeks, without making a single error. There are, of course, times when they are out of order and make errors. It is as much a part of programming and coding to determine when the high-speed computers are calculating correctly as it is to prepare a problem for them. This discussion of speeds and errors suggests that the high-speed arithmetic unit should be coupled with a device which can keep track of the answers as quickly and as efficiently as the arithmetic unit produces them. The high-speed arithmetic units are expensive and should be kept in continuous operation in order to be justified economically. The device which is usually coupled to the arithmetic unit is called the *memory* of the high-speed computer. Memories are of various capacities and designs as will be discussed later. For the present, we will only say that the fastest types of memories that are in use have random accessibility and can hold (remember) somewhere from one thousand to one hundred thousand 10-decimal-digit numbers. A *random access* memory is one which has equal availability for all numbers stored in it. That is, the time for referring to n numbers in the memory depends only on n and not on the reference sequence. Reference to a random sequence is as fast as any other sequence.

The word *word* has become a generic term in the computer field meaning a set of digits, usually of fixed length, e.g., 10-decimal digits and a sign digit. Thus, we may refer to the "number of words" a memory holds, or to the "number of words" the computer operates on in a given time, etc.

The question of whether problems that require high-speed computation and storage of large amounts of information actually exist is only natural. The numerical solution of n simultaneous linear algebraic equations requires approximately n^3 arithmetic operations. The time-dependent solution of the hydrodynamical equations in two space variables, where one desires a solution at only ten positions in each direction at each time step, may require 1,000 time steps to carry the solution to the point of interest. In such a problem one is solving 100 simultaneous equations at each time step, or one is performing of the order of 10^9 operations for the complete problem. At 15 microseconds per operation this problem would take 1.5×10^4 seconds, or approximately 4 hours

of computing time. If the same problem were performed without error on a desk calculator, it would take 4×10^9 seconds, or approximately 130 years. Thus, we have not only demonstrated a problem of ample size, but one that would have to be left uncomputed if it were not for the high-speed computer. We should also point out that these high-speed computers vary in speeds as well as size of memory, and that a computer which runs at a speed 1,000 times slower than the stated speed could still produce the answers to the preceding problem in a half-year's time. Even such a slow high-speed computer—usually referred to as a *medium-speed* computer—is too fast to be coupled with a human memory.

Now that we have justified the need for programming and coding of automatic digital high-speed computers, let us turn to the problem of learning some of the basic rules for this science. One method is to state these rules and then try to program and code problems based on the rules. Another is to explain the need for the rules by tying them to the operation of the computer or to the logic of calculating problems. Still another method, the approach which we will follow, is that of constructing a computer—in words and pictures, but not physically—and, in so doing, finding some of the computer requirements placed on programming and coding of existing computers.

1-2. Some Basic Concepts in the Design of a Computer

As we indicated in the introduction, a high-speed digital computer consists of an arithmetic unit coupled with, at best, a random access memory. In block diagram we might show this relationship as in Fig. 1-2-1 where the arrows indicate the flow of information. Let us assume for the purpose of this example that the memory will hold 1,000 words, each word containing 10-decimal digits and sign. The sign digit of many computers assumes only two values used to represent + and −. Other computers allow additional values and other interpretations for this digit. In our description we will allow the sign digit when used in an order word to take on the digits 0 through 9 just as any other digit in the word. When used with a data word, it will not only indicate whether the word is positive or negative, but will also indicate whether or not the word contains alphabetical characters. We will designate the digits of a word as

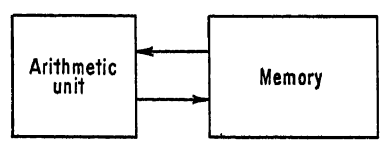


FIG. 1-2-1. Coupling of arithmetic unit and memory.

$$sd_0d_1d_2d_3d_4d_5d_6d_7d_8d_9 \qquad (1\text{-}2\text{-}1)$$

The words shall be stored in units of the memory which are called *cells* or *storage registers*. Each of the 1,000 memory cells shall be designated by a 3-digit serial number which is called the *address* or *location* of a cell. The 1,000 addresses are 000, 001, 002, . . . , 999. It is important to recognize the distinction between the address, or location, of a cell and the word stored in that cell. The address specifies the cell location (location is frequently used as a synonym for address). The address may be considered as the coordinates for the location of a memory cell. In Table 1-2-1, illustrating the addresses and contents for 60 cells of memory, cells addressed 000 through 019 contain orders (commands for the computer), that is, the information stored in these cells is to be used to sequence the operations of the computer. The contents of a memory cell may be recalled without being destroyed; however, the storing of a new word in a cell destroys its previous contents.

TABLE 1-2-1. A Partial Memory Chart Showing Address and Contents of Memory Cells

Digits of address										
First and second	Third									
	0	1	2	3	4	5	6	7	8	9
00	Order	Order	Order	Order	Order	Order	Order	Order	Order	Order
01	Order	Order	Order	Order	Order	Order	Order	Order	Order	Order
02	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data
03	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data
04	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data
05	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data

We next consider automatic control of the arithmetic unit and memory. Introducing more symbolism, we let M designate the address of a cell in the memory and m the word located at that address, i.e., stored in that cell. In the remainder of this text we will, generally, use capital letters for addresses and small letters for the words located at those addresses. However, we will often find that the alphabet is not sufficient to describe the variables of a problem, or that we have need for a capital letter to represent something other than an address, or that a small letter is needed to represent something other than the contents of a memory cell. Thus, we will also use $M(x)$ to represent the address of the word x and

$C(X)$, or just (X), to represent the contents of the cell addressed X; i.e., $X = M(x)$ and $x = (X) = C(X)$. The addresses of the memory cells of the model computer being considered require, at most, three decimal digits. For convenience, the address will always be considered to have three digits where zeros are filled in the high-order digits of the address when necessary. This is a requirement of the computer circuitry for the selection of a particular cell. Arithmetic operations usually operate on two numbers to produce a third. These operations may be described in terms of three addresses (M_1, M_2, and M_3) and an instruction code. For example, the memory might send the two words m_1 and m_2, the addresses of which are M_1 and M_2, to the arithmetic unit. The arithmetic unit is then instructed to perform one of the arithmetic operations on m_1 and m_2. This produces a result m_3, which is stored then in memory cell M_3. Let

M_1 be designated by the digits $d_1d_2d_3$
M_2 be designated by the digits $d_4d_5d_6$
M_3 be designated by the digits $d_7d_8d_9$

of the expression (1-2-1), then we will have left the digit d_0 and the sign, s, to describe the arithmetic operation or computer instruction (or computer operation), which is designated by I,

I is designated by the digits sd_0

Thus, digital codes for orders can be stored in the memory of the computer as follows:

$$I\,M_1\,M_2\,M_3 \tag{1-2-2}$$

For the arithmetic operations the expression of (1-2-2) may be interpreted as

$$I(m_1,m_2) = m_3 \tag{1-2-3}$$

the operands m_1 and m_2 when operated on by the instruction I produce a result m_3 which is then stored in the memory. As we proceed we will find orders involving processes other than the arithmetic operations for which Eq. (1-2-3) will not hold. It is evident that either an order or a number could be stored in a memory cell. The interpretation of the contents of a cell will depend on the context for which it is used in a computation.

If there is a sufficient number of digits in a word, then such a word may be used to store more than one instruction and the associated addresses. For example, if there are 21 decimal digits, d_0 through d_{20}, and a sign digit s, an order word might consist of

$$I \quad M_1 \quad M_2 \quad M_3 \qquad I' \quad M_1' \quad M_2' \quad M_3'$$

where I' is designated by the digits d_{10} and d_{11}. Such an order word is said to consist of two commands, $IM_1M_2M_3$ and $I'M'_1M'_2M'_3$. Most of the computers we will discuss will have a single command per order word, and "command" and "order" will be used interchangeably.

Now that the possibility of storing order words in the memory as well as words of data to be operated upon has been demonstrated, the need is apparent for adding another unit to the computer. This unit will be used to decipher orders and to instruct the memory of the operands to be sent to the arithmetic unit, to instruct the arithmetic unit of the operation to be performed, and to locate the result in the memory. This additional unit is commonly referred to as the "control unit." A part of the control unit is the *control register* (abbreviated CR), which is a storage register for holding the order being deciphered. If we draw a box for the control register, as in Fig. 1-2-2, then we see how each part of an order word can be separated and used for its particular purpose. In addition to the control register, there must be a device that tells which order is to be sent from the memory to the control register. This device, the *control counter* (CC), is a 3-digit counter which increases by 1 in its low-order position each time an order is performed. Modification of this sequencing can be accomplished only by an appropriate instruction. The CC and CR are parts of the *control unit* (CU) the function of which is to sequence and coordinate the activity of the computer.

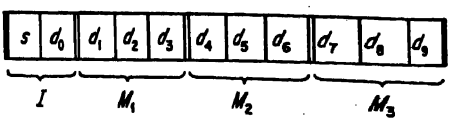


FIG. 1-2-2. Order word of a three-address computer.

It is instructive to examine how the control unit might sequence the operation of the computer. The following sequence starts with the transfer of an order from the memory to the CU.

1. CU notifies the computer to prepare to read the order from the memory cell, the address of which is in CC.
2. CU notifies memory to send order to CR.†
3. CU increases CC by one.
4. CU notifies the arithmetic unit to prepare for the instruction I in CR.
5. CU notifies the memory to prepare to read from M_1 (in CR).
6. CU notifies memory to send word (the address of which is M_1) to arithmetic unit.†

† As mentioned earlier this transmission does not destroy the contents of the memory cell; i.e., words are read from the memory in a nondestructive manner.

7. CU notifies the memory to prepare to read from M_2 (in CR).
8. CU notifies memory to send word (the address of which is M_2) to arithmetic unit.†
9. CU notifies arithmetic unit to execute the arithmetic operation (which was indicated by I).
10. CU notifies the memory to prepare to read into M_3 (still in CR).
11. Arithmetic unit notifies CU that the arithmetic operation is completed. (This step is necessary because of the different operation times for the arithmetic operations.)
12. CU notifies arithmetic unit to send result to memory cell addressed M_3.‡
13. CU returns to step 1 to automatically start the next command.

Except for getting words in and out of the computer, and for provisions for human control over the computer, we have covered the basic units of our model computer. The equipments used to get orders and data in and out of the computer are referred to as the *input* and *output* units. Sometimes these units are combined and are referred to as the *input-output* unit. Again, like most other units of the computer, the input and output units function in various ways. For the present, we will assume that the input and output unit communicates with the memory in a manner which allows us to load words in the memory, and the output unit allows the memory to be read out in some intelligible form, say, through a typewriter.

The communication between human and computer is accomplished through the *console* of the computer. Among the possible functions that can be performed at the console, the human should be able to start and stop the computer and have some means of manually setting the control register and control counter. For speed and error reduction operations performed by humans should be kept to a minimum.

Figure 1-2-1 may now be expanded and our computer considered to be basically that of Fig. 1-2-3. Figure 1-2-3 is by no means complete; but when we consider the arithmetic operations, it indicates many of the basic principles of the computer. For example, it is desirable to have a means of input and output located at the console so that the human can observe the progress of the problem being computed, so that programs can be initially tested, so that the computer can be stopped when errors are detected, etc. The solid lines show the paths along which infor-

† See footnote on page 8.

‡ This operation replaces the previous contents of cell M_3 by the result for this command.

mation flows, and the dotted lines show the control of one unit over another. The parts of the computer not shown include all of the switching circuits that select the information flow paths, the function matrix that deciphers the instruction portion of the orders, and the power supply that provides the energy at the proper rates to run the various units of the computer. These parts, which we only mention in passing, are a substantial part of the computer's physical size.

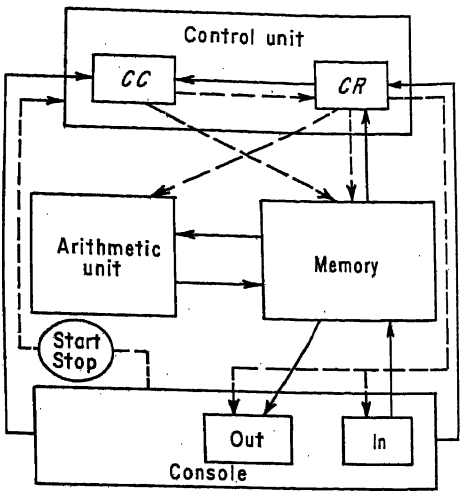


FIG. 1-2-3. Basic units of a three-address computer.

1-3. Examples of Orders

In this section, we shall choose a small number of orders which allow the solution of a wide variety of problems. For the instruction part (I) of the order we shall use the following eight symbols: ADD (add), SUB (subtract), MLR (multiply), DVR (divide), TRA (transfer conditionally), LOD (load), PRT (print), and STP (stop). In the description of the orders which follows, we will use the symbol $\Rightarrow$ to mean "implies."

ADD $M_1M_2M_3 \Rightarrow m_1 + m_2 = m_3$; i.e., add the word addressed M_1 to the word addressed M_2 and store the sum in memory location M_3.

SUB $M_1M_2M_3 \Rightarrow m_1 - m_2 = m_3$; i.e., subtract the word addressed M_2 from the word addressed M_1 and store the difference in memory location M_3.

MLR $M_1M_2M_3 \Rightarrow m_1 \times m_2 = m_3$; i.e., multiply the word addressed M_1 by the word addressed M_2 and store the computer product in memory location M_3.

DVR $M_1M_2M_3 \Rightarrow m_1 \div m_2 = m_3$; i.e., divide the word addressed M_1 by the word addressed M_2 and store the computer quotient in memory location M_3.

TRA $M_1M_2M_3 \Rightarrow$ if $m_1 \geq m_2$, change CC to M_3; i.e., if the word addressed M_1 is algebraically greater than or equal to the word addressed M_2, change the computer control counter contents to the digits $d_7d_8d_9$; i.e., to M_3.

LOD $M_1M_2M_3$ $\Rightarrow$ load M_1 words from input unit into memory starting at address M_2 and, when finished loading, change CC to M_3.

PRT $M_1M_2M_3$ $\Rightarrow$ print from memory M_1 words starting at address M_2 and, when finished printing, change CC to M_3.

STP $M_1M_2M_3$ $\Rightarrow$ stop operation of computer. Ignore M_2. If M_1 is 000, ignore M_3; and, if M_1 is 001, change CC to M_3.

Now let us examine these orders more closely and see what they imply about our model computer. For example, the instructions MLR and DVR say that the computer product and computer quotient respectively are stored in the memory. So far, our computer has been considered to behave algebraically. That is, we have considered that arithmetic operations on two words produce one-word results, and no consideration has been given to the placement of the decimal point. We have assumed a word length of ten digits and sign. However, the multiplication of two 10-digit words generally produces a 20-digit product. We will assume that the product is approximated by a 10-digit word where the last digit to the right has been rounded according to the digits dropped from the 20-digit product. Similarly, in division the quotient is assumed to be a 10-digit word with the last digit to the right rounded according to the remainder. Finally, in addition and subtraction carries beyond the most significant digit are neglected.[1]

Next, if we review the 13 CU steps (the subinstructions of the computer), we see that these steps represent the behavior of the computer when performing arithmetic operations. However, the list of orders includes not only the four arithmetic orders ADD, SUB, MLR, and DVR, but also a decision order TRA, a control order STP, and two input-output orders LOD and PRT. For the control orders, M_1 is not used as a memory address, but is used as a subinstruction of I. Thus, the operation of the computer with respect to the CU steps must be generalized. This may be accomplished as follows. Replace step 4 by

4. CU notifies the relevant parts of the computer to prepare for the instruction I in CR.

[1] If an intermediate result unknowingly exceeds these representations for results, then a gross error would have occurred. In all computations it is of course important that the programmer have an intimate knowledge of the procedures he uses and the nature of the results. The computer is frequently a useful tool for gaining this knowledge. See G. E. Forsythe, The Role of Numerical Analysis in an Undergraduate Program, *Amer. Math Monthly*, vol. 66, no. 8, pp. 651–662, October, 1959.

At this point the CU chooses between two sequences. If I is an arithmetic or decision instruction, then the computer carries out steps 5, 6, 7, and 8 as previously stated. If I is arithmetic, then the computer continues as before. If I is TRA, then the steps from 9 through 12 are replaced by:

9. CU notifies arithmetic unit to subtract (i.e., compute $m_1 - m_2$).
10. Arithmetic unit notifies CU it has completed operation.
11. If sign of difference in arithmetic unit is minus, CU skips to step 1. If sign is positive, CU goes to step 12. Here, we assume that the sign of a zero difference is plus.
12. CU notifies CR to send M_3 to CC.

If, at step 4, I had been a control or input-output instruction, then the following steps would replace 4 through 12 (numbering of steps refers to CU timing already chosen, and missing numbers imply that CU skips those steps):

4. CU notifies relevant parts of computer to prepare for instruction IM_1 in CR.
7. CU notifies memory of address M_2 in CR.
9. CU notifies CR to send M_3 to CC if required.
11. CU notifies relevant parts of computer to execute IM_1 instruction and, when finished, CU skips to step 13.

We will further assume that pushing the stop button on the console or a STP instruction stops the computer after CU step 13, but before CU step 1, and that the start button on the console starts the computer either at CU step 1 or CU step 4 as will be explained later. Also, we will assume that when a word is sent to a particular location in memory, any previous information in that location is erased when the new word is stored; but if a word is read from memory, it is not erased in its original location by the reading process.

The initial loading of the computer memory will now be discussed. A LOD instruction must be in CR in order for the computer to load information from input unit into memory. This may be accomplished by having a special switch, referred to as the *mode of operation switch*, on the console which permits a choice between continuous operation and single command operation and by being able to set CR and CC from the console. When the switch is set to single command operation and the

start button is pushed, the computer performs only the command in CR, i.e., the computer starts at step 4 and stops after step 13. When the switch is set to continuous operation and the start button is pushed, this computer operation starts with step 1 of the sequence of computer operations. Thus, when starting the computer in the continuous mode of operation, CC must be preset. It should be set to the address of the command to be performed.

Having completed these preliminaries, in the remaining sections of this chapter we introduce two simple problems and some of the programming and coding concepts for these problems, saving the details for Chap. 3.

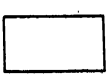
1-4. Examples of Programming Computer Problems

A simple but typical computer problem is that of forming the sum of a set of numbers. Let the set contain n numbers which we designate by $a_1, a_2, \ldots, a_n$. The desired sum is

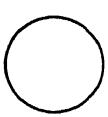
$$s = \sum_{j=1}^{n} a_j = a_1 + a_2 + \cdots + a_n \qquad (1\text{-}4\text{-}1)$$

In programming such a problem for a computer, the object is to write a set of orders which will be modified and repeated by the computer until the n numbers have been totaled.

Next, let us draw a "road map" or diagram for this problem. For this purpose the following symbols are used:

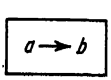


Boxes are used to separate the problem into its basic parts.

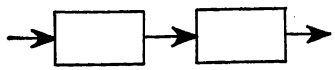


Circles are used to indicate where the problem begins (starts) and ends (stops).

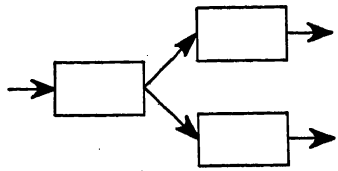
Arrows indicate the sequence of the parts of the problem.



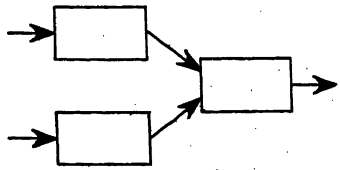
An arrow in a box indicates that the quantity at the tail of the arrow replaces the quantity at the head of the arrow. In this case the new value of b is a.



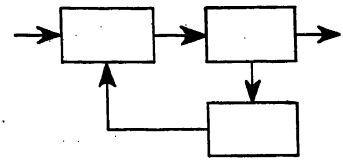
Indicates a predetermined computational sequence.



Indicates an alternative computational sequence.



Indicates a merging computational sequence.



Indicates an iterative computational sequence (composed of predetermined, alternative, and merging sequences).

The road map or *flow diagram* for calculating the sum in Eq. (1-4-1) is shown in Fig. 1-4-1. Flow diagrams are introduced to aid in the transition from the mathematical statement of a problem to the coding for the problem. This flow diagram has the following interpretation. Initially, the sum s is set to zero and the subscript j is set to one in box 0. In the first pass through box 1, a_j, where $j = 1$, is added to s. Thus, on leaving box 1 for the first time, $s = a_1$. In box 2, the subscript is increased by one; and in box 3, a test is performed to see if $n \geq j$. If $n \geq j$, then the routine is to return to box 1, and the next a_j of the set is added to s. When $n = j$, the routine is again returned to box 1 and a_n is added to s; one is added to j in box 2; and, in box 3, $j = n + 1$. Therefore, the routine proceeds to box 4, and the sum is listed at an output unit.

There are several observations to make concerning the flow diagram of Fig. 1-4-1. The first is that a box, such as box 3, in which a decision occurs (in which an inequality is written), must have two paths leading from it. One path is marked *yes* to show that along this path the inequality holds; the other path is marked *no* to show that the inequality does not hold along that path. A question mark is included in the box in which the decision occurs to emphasize that a question is being asked rather than that a fact is being stated. Thus the contents of box 3 should

be read, "Is $n \geq j$?" The second is that any box may have more than one path leading to it; e.g., box 1. Finally, we have numbered the boxes. We will make use of this numbering in the coding of the problem.

Another problem is that of finding the square root of a positive number. It illustrates many phases of programming and coding. The method to be used for finding the square root is that commonly referred to as *Newton's method.* This method is chosen because it is an iterative method; that is, the same formula is used repeatedly. Iterative methods frequently reduce the coding. The statement of the problem follows:

Find $x = \sqrt{y}$, given $y > 0$, by repeated use of the formula

$$x_{i+1} = \frac{1}{2}\left(\frac{y}{x_i} + x_i\right) \qquad (1\text{-}4\text{-}2)$$

where $x_0 = y$ (1-4-3)

and $x = x_{i+1}$ (1-4-4)

if $\epsilon \geq (x_{i+1} - x_i)^2$ (1-4-5)

$\epsilon > 0$ is a criterion for the desired accuracy of x.

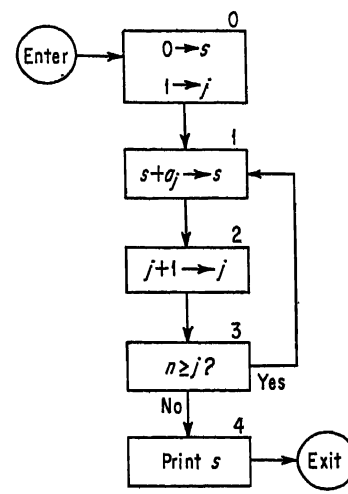


FIG. 1-4-1. Flow diagram for $s = \sum_{j=1}^{n} a_j$.

In words, the method of this problem is to let y be an initial approximation for x (i.e., $x_0 = y$, $i = 0$). Next, Eq. (1-4-2) is used to obtain an improved approximation, x_{i+1}, for x, and then the square of the difference $(x_{i+1} - x_i)^2$ is checked by Eq. (1-4-5) to see if the improved approximation is adequate. If Eq. (1-4-5) is satisfied, $x = \sqrt{y}$ is obtained from Eq. (1-4-4); if Eq. (1-4-5) is not satisfied, x_i is replaced by x_{i+1}, and Eq. (1-4-2) is again used to obtain another improved approximation. By setting $x_{i+1} = x_i = x$, Eq. (1-4-2) becomes $y = x^2$ or $x = \sqrt{y}$. The smaller ϵ is chosen, the better the approximation obtained for x. Newton's square-root routine, Eqs. (1-4-2) through (1-4-5), is shown in Fig. 1-4-2.

Comparing the flow diagram of Fig. 1-4-2 with the problem, we see that it describes the mathematical method for the problem. From the entry circle and problem loading in box 0 we proceed to box 1 where x_i is replaced by y, the initial approximation for x. From box 1 we go to box 2 in which an improved approximation, x_{i+1}, is obtained for x. Next, in box 3, the difference between the old and new approximations for x is

formed; and, in box 4, the square of the difference is formed to obtain a positive measure z^2 of this difference. In box 5, ϵ is compared with z^2 and if $\epsilon \geq z^2$, then x_{i+1} is substituted for $x = \sqrt{y}$ in box 7, and the problem is completed. If $\epsilon < z^2$, then x_i is replaced by x_{i+1} in box 6, and we return to box 2 to obtain another improvement for the approximation of x (*cf.* Sec. 10-7).

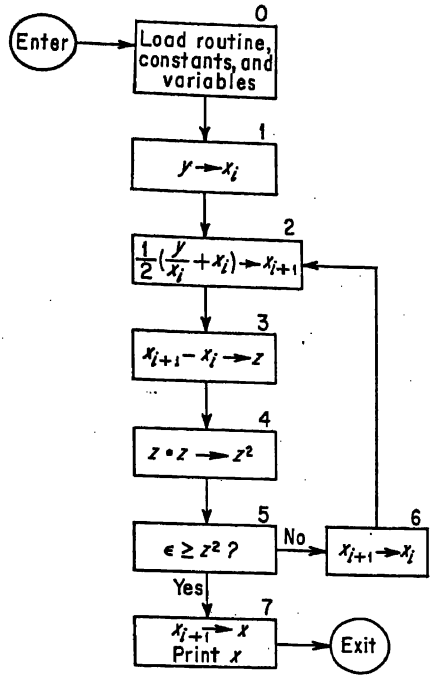


FIG. 1-4-2. Flow diagram for $x = \sqrt{y}$.

1-5. Coding of Example Computer Problems

A coding sheet for the summation problem, Eq. (1-4-1), is presented in Table 1-5-1. At first we stress the filling of the columns titled Box; I, M_1, M_2, and M_3 of Order Symbols; and Branch to Box. To fill in these columns of the coding sheet, we place the flow diagram of Fig. 1-4-1 in front of us. Since the first operation of box 0 is to set $s = 0$, we accomplish this by adding zero to zero and storing the sum in S, the address of the memory cell which contains s. Thus, in column I, we place ADD, the symbol for the addition instruction; in each of the columns M_1 and M_2, we place a zero which we interpret as the address of the number 0 stored in the memory (if we had meant that the addresses M_1 and M_2 were zero, we would have written 000); and in column M_3, we write S. Next we fill in the column headed Box. In this column the box number of the flow diagram is given to the left of the decimal point, and the command associated with the box is indicated by the number to the right of the decimal point. Similarly, to set $j = 1$, one is added to zero, and the sum is stored in J, the address of the memory cell in which j is stored. Again, columns I, M_1, M_2, M_3, and box are filled in. Similarly, these five columns are filled in for the remaining boxes of the flow diagram. In the command for box 1, a_1 instead of a_j is added to s; and in the commands for box 2, every time j is incremented by 1, the address A_1 is increased by one by adding the word d (in this case

$d = +0\ 000\ 001\ 000$) to the command, designated by $C_{1.0}$, for box 1. This changes the command so that it no longer refers to A_1 but to A_2. This procedure is repeated each cycle until A_n is reached. We have assumed that the a_j are stored in consecutive memory cells. The command for box 3 is that for determining when the sum is completed. This occurs when $j = n + 1$. Whenever $n \geq j$, the routine returns to the

TABLE 1-5-1. Coding Sheet for $s = \sum_{j=1}^{n} a_j$

Box	Order symbol				Branch	Memory location	Order code			
	I	M_1	M_2	M_3			I	M_1	M_2	M_3
START										
0.0	ADD	0	0	S		100	01	200	200	201
0.1	ADD	1	0	J		101	01	202	200	203
1.0	ADD	S	A_1	S		102	01	201	300	201
2.0	ADD	1	J	J		103	01	202	203	203
2.1	ADD	D	$C_{1.0}$	$C_{1.0}$		104	01	204	102	102
3.0	TRA	N	J		1.0	105	22	205	203	102
4.0	PRT	001	S	C_{EXIT}		106	42	001	201	107
EXIT	STP	000	000	000		107	00	000	000	000
	←		0	→		200	+0	000	000	000
	←		s	→		201	←		s	→
	←		1	→		202	+0	000	000	001
	←		j	→		203	←		j	→
	←		d	→		204	+0	000	001	000
	←		n	→		205	+0	000	000	$n_7n_8n_9$
	←		a_1	→		300	←		a_1	→
	←		a_2	→		301	←		a_2	→
			⋮						⋮	
	←		a_n	→		300 + $n-1$	←		a_n	→

command $C_{1.0}$, and another a_j is added to the sum. Thus, in the column headed Branch, we indicate the command to which the routine returns and leave column M_3 blank. In this manner we have a positive indication on the coding sheet of the existence of a *loop* in the flow diagram. A loop is a closed path which indicates a repetition of a part of the flow diagram.

After the order symbols for the commands are completed, a list of the constants and variables, except for the a_j, is compiled from the commands; and, then, the a_j are listed. Next, the memory locations for the orders, constants, and variables are assigned. Here, we have chosen addresses

for the orders starting at 100, addresses for the constants and variables starting at 200, and addresses for the a_j starting at 300 with the assumption that $A_n \leq 999$.

Now we are in a position to fill in columns M_1, M_2, and M_3 of Order Code, which is done as shown in the coding sheet. Before column I of Order Code can be filled in, we must assign the digital codes for s and d_0 for the commands given in Sec. 1-3. Let the assignments be: ADD = 01, SUB = 02, MLR = 03, DVR = 05, TRA = 22, LOD = 41, PRT = 42, and STP = 00. The particular digit assignments will become clearer as the command list for the computer is expanded to demonstrate other facets of programming and coding. Column I of Order Code is filled in as shown in the coding sheet.

At this point, we note that we have instructed the computer to add a number to a command. We will assume that, when either M_1 or M_2 is the address of an order word, our computer will add only the digits d_1 through d_9 and that the digits s and d_0 of the sum will be the same as those for the order word.

Finally, the actual digits of the constants, such as 0, 1, d, n, and the a_j are listed. The digits n_7, n_8, n_9 of n are those that may be nonzero since we have assumed that there are, at most, 700 a_j's ($A_1 = 300$ and $A_n \leq 999$).

The coding sheet of Table 1-5-1 is by no means presented as being the optimum coding for the problem. Also, no provisions have been made for loading the routine into the memory of the computer. However, should the routine be in memory, we can see that by setting the CC to 100 and pressing the start button of the computer, the sum will be formed. Another objection is that the problem cannot be repeated without reloading the routine, for if we were to reset the CC to 100 and to start the problem a second time, the computer would form the sum of $C(300 + n)$ through $C(300 + 2n - 1)$ which are not involved in the problem. Finally, since A_1 and j are both incremented, commands 0.1 and 2.0 could have been neglected and command 3.0 changed to a test on A_j. A more economical coding of this problem using this suggestion will be developed in later chapters. The purpose of this section is to demonstrate how one proceeds from the problem to the flow diagram to the coding sheet.

In addition to the flow diagram and coding sheets, the programmer often finds constructing a *memory chart* useful in assigning addresses for orders and data and in checking the consistency of these assignments. A *static memory* (or *storage*) *chart* is a table indicating the initial assignment of memory cells for the problem. Table 1-2-1 indicated such a

chart; however, it was developed to show how the address of a cell might be considered as coordinates for the location of that cell. Table 1-5-2 is a static memory chart for the problem coded in Table 1-5-1. Also indicated in this chart is the possible location of a load and print routine. Memory charts which show the progress of the computation are referred to as *dynamic memory charts.* These are usually constructed from the

TABLE 1-5-2. Static Memory Chart for Coding of Table 1-5-1

Digits of address																					
First and second	Third										First and second	Third									
	0	1	2	3	4	5	6	7	8	9		0	1	2	3	4	5	6	7	8	9
0 0	← Load and print										0 1	routine →									
0 2	← Load and print										0 3	routine →									
0 4	← Load and print										0 5	routine →									
0 6											0 7										
0 8											0 9										
1 0	← Orders →										1 1										
1 2											1 3										
1 4											1 5										
1 6											1 7										
1 8											1 9										
2 0	0	*s*	1	*j*	*d*	*n*					2 1										
2 2											2 3										
2 4											2 5										
2 6											2 7										
2 8											2 9										
3 0	← a_j										3 1	→									
3 2	← a_j										3 3	→									
3 4	← a_j										3 5	→									
3 6	← etc.										3 7	→									
3 8											3 9										

coding sheets and are used to check that no information is destroyed before it is used. Such charts are especially useful in checking routines for single-address and two-address computers where the contents of the accumulator, which may also be destroyed, is also displayed (*cf.* Sec. 3-5).

Now let us turn to our second example, that of finding the square root of a positive number. Again, we place the flow diagram, Fig. 1-4-2, in front of us and proceed to fill in columns I, M_1, M_2, and M_3 of Order Symbol in Table 1-5-3. To start the problem, we need a load order; so

we place LOD in the I column. Columns M_1, M_2, and M_3 are not filled in at this time since M_1 represents the number of words to be stored in the memory; M_2 represents the address of the first word in the memory; and M_3 is the setting of the CC for the first order to be performed. Since the first order occurs in box 1, 1.0 is placed in the Branch-to-Box column

TABLE 1-5-3. Coding Sheet for $x = \sqrt{y}$

Box	Order symbol				Branch to box	Memory location	Order code			
	I	M_1	M_2	M_3			I	M_1	M_2	M_3
	LOD	(022)	(900)	(900)	1.0	...	41	022	900	900
1.0	ADD	Y	0	X_i		900	01	912	919	914
2.0	DVR	Y	X_i	T_1		901	05	912	914	917
2.1	ADD	T_1	X_i	T_1		902	01	917	914	917
2.2	DVR	T_1	2	X_{i+1}		903	05	917	921	915
3.0	SUB	X_{i+1}	X_i	Z		904	02	915	914	916
4.0	MLR	Z	Z	T_2		905	03	916	916	918
5.0	TRA	E	T_2	$C_{7.0}$	7.0	906	22	920	918	(909)
6.0	ADD	X_{i+1}	0	X_i		907	01	915	919	914
6.1	TRA	2	0	$C_{2.0}$	2.0	908	22	921	919	(901)
7.0	ADD	X_{i+1}	0	X		909	01	915	919	913
7.1	PRT	001	X	$C_{7.2}$	7.2	910	42	001	913	(911)
7.2	STP	...	...	...		911	00	000	000	000
	Constants and variables									
	← y →					912	←		y	→
	← x →					913	←		x	→
	← x_i →					914	←		x_i	→
	← x_{i+1} →					915	←		x_{i+1}	→
	← z →					916	←		z	→
	← t_1 →					917	←		t_1	→
	← t_2 →					918	←		t_2	→
	← 0 →					919	←		0	→
	← ϵ →					920	←		ϵ	→
	← 2 →					921	←		2	→

to remind us that M_3 will have the address of the first order of box 1. In the next row of the coding sheet 1.0 is placed in the Box column, ADD is placed in column I, Y is placed in column M_1, 0 is placed in column M_2, and X_i is placed in column M_3. This order, then, substitutes y for x_i. Similarly, the coding sheet is filled out for boxes 2, 3, 4, 5, 6, and 7. Since box 6 leads back to box 2 in the flow diagram, the order 6.1 on the coding sheet must be one that will reset the CC. A TRA order was chosen for this purpose. The constants are chosen so

that the transfer condition is always satisfied, i.e., the control unit is always changed to M_3. The addresses Z, T_1, and T_2 are for *temporary* storage cells in the memory; i.e., where intermediate results are stored. So that the answer, $x = \sqrt{y}$, will be known, the print order 7.1 is added; and, of course, it is desired to stop the computer—order 7.2—when computation is completed. Below the coding of orders, the constants and variables are listed.

Let the program for the square-root problem be stored in the memory starting at memory address 900. Since the load order need not go into the memory, the first memory location (900) is assigned to the order 1.0. The column marked Memory Location is now filled in as indicated. When this column is completed, we may return to the columns of Order Symbol and fill in the blank spaces. For example, in the load order, $M_1 = 022$ since there are 22 words, addressed 900 through 921, to be stored in the memory. M_2 is the location of the first word, i.e., $M_2 = 900$; and M_3 is the address of order 1.0, i.e., $M_3 = 900$. Similarly, the M_3 portion of orders 5.0, 6.1, and 7.1 are completed. These numbers have been circled on the coding sheet to show that they are filled after the memory assignments are made.

The remaining columns may now be filled by placing in column I of Order Code the code numbers for the instruction in column I of Order Symbol and by placing in columns M_1, M_2, and M_3 in Order Code the numerical addresses corresponding to M_1, M_2, and M_3 in Order Symbol. These number codes and addresses are filled in between the double lines of the coding sheet, and it is this information which is actually placed in the input unit of the computer. Thus, the information placed in the input unit may be traced to the coding sheet, the coding sheet to the flow diagram, and the flow diagram to the problem. Each step in the programming and coding of the problem has been set up to reduce the possibility of making errors and to facilitate the location of an error should it occur. The load command at the top of the coding sheet may now be placed in the CR and the computer set to carry out this instruction. When the routine has been entered in the memory, i.e., the LOD instruction in CR completed, the computer will automatically proceed to the first command, address 900, and thence calculate the square root of y.

1-6. Two-address Decimal Computer

If our model computer had a memory of between 1,001 and 10,000 words, each of ten digits and sign, then the minimum number of digits required for the address part of an order word is four. Each order word,

then, could contain only two such addresses and an instruction of two digits and sign. For such a computer the control register may be subdivided as shown in Fig. 1-6-1. Within this order structure, the computer can no longer follow the three-address logic for arithmetic operations given by Eq. (1-2-3). Parts of the computer and the computer's order list will have to be reconsidered. In Fig. 1-6-1 we have indicated that M_1 is generally considered as the memory address of an operand. By considering M_2 as the address of the next order to be performed, the control counter (CC) may be replaced by an *address register* (AR) which notifies the memory of the address of the next order. Also, the second address part of CR could be used as an address register. An address register is somewhat simpler to build electronically than a control counter, and the above choice of function of M_2 is dictated by a balance of programming and engineering considerations. In the arithmetic unit the results of arithmetic operations are stored in a register commonly referred to as the *accumulator*. The abbreviation for the accumulator is A. As will be seen in Chaps. 2 and 3, the accumulator is divided into two registers, the *upper accumulator*, A_U, and the *lower accumulator*, A_L, each of which contains one word and may be used in addition or subtraction. However, the full accumulator is used in multiplication, division, and other orders. The basic units and flow lines for information of the new computer are shown in Fig. 1-6-2. This computer is referred to as a *two-address computer*. The terms *modified single address*, *one and one-half address*, and *modified two-address* are also used in the literature when referring to this type of computer. The two-address computer we will use for coding purposes in this text will be slightly different from the one just described in that it will have a control counter instead of an address register. Furthermore, we assume that if the order address is left blank, the computer reads this as 0000 and inter-

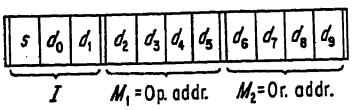


FIG. 1-6-1. Order word for a two-address computer.

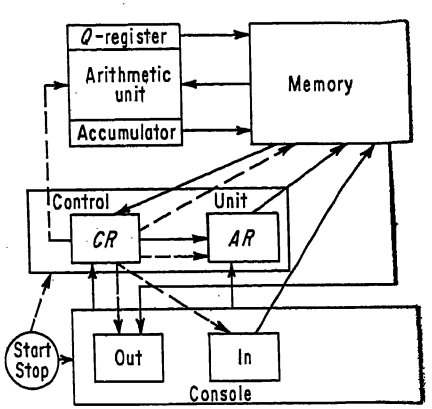


FIG. 1-6-2. Basic units of a two-address computer.

prets it to mean that the next order is taken sequentially. However, if the order address is nonzero, then the next order is taken from that memory location. This system prohibits the location of an order word at the memory location 0000. This location is used for the permanent storage of zero, a frequently used constant.

The instruction consists of the sign and digits d_0 and d_1, and we need only four characters for the sign digit, + and − to be used with order words and numerical data words and a special plus and minus sign, ⊕ and ⊖, for data words containing alphanumerical characters.

A partial list of orders which will be used with this computer follows:

CAU M_1M_2 (Clear Add Upper) $\Rightarrow m_1 \rightarrow a_U$, where a_U is the contents of A_U, $0 \rightarrow a_L$. (Clear the entire accumulator to zero and then send m_1 to A_U.) Next order taken from memory location M_2 if $M_2 \neq 0$.

HAU M_1M_2 (Hold Add Upper) $\Rightarrow a + m_1 \rightarrow a$, where a is the contents of A. (Here, m_1 is being added to A in the A_U position, and zeros are being added to A_L. If m_1 were added to A in the A_L position, it would be indicated by $a + 10^{-10} m_1 \rightarrow a$.) Next order taken from memory location M_2 if $M_2 \neq 0$.

HSU M_1M_2 (Hold Subtract Upper) $\Rightarrow a - m_1 \rightarrow a$; next order taken from memory location M_2 if $M_2 \neq 0$.

CAQ M_1M_2 (Clear Add Q register) $\Rightarrow m_1 \rightarrow q$, where q is the contents of the Q register. (The Q register is a special register in the arithmetic unit used with multiplication and division commands.) Next order taken from memory location M_2 if $M_2 \neq 0$.

MLR M_1M_2 (MuLtiply Round) $\Rightarrow qm_1 \rightarrow a$, where the rounded product appears in A_U. Next order taken from memory location M_2 if $M_2 \neq 0$.

DVR M_1M_2 (DiVide Round) $\Rightarrow a/m_1 \rightarrow q$, where q is the rounded quotient; next order taken from memory location M_2 if $M_2 \neq 0$.

STU M_1M_2 (STore Upper) $\Rightarrow a_U \rightarrow m_1$; next order taken from memory location M_2 if $M_2 \neq 0$.

STQ M_1M_2 (STore Q register) $\Rightarrow q \rightarrow m_1$; next order taken from memory location M_2 if $M_2 \neq 0$.

TAP M_1M_2 (Transfer Accumulator Positive) $\Rightarrow$ If $a > 0$, next order taken from M_1; if $a \leq 0$, next order taken from M_2 if $M_2 \neq 0$.

UCD M_1M_2 (UnConDitional transfer) $\Rightarrow$ Next order taken from memory location M_2. Ignore M_1.

LOD M_1M_2 (LOaD) $\Rightarrow$ Load M_1 words from input unit into consecutive memory cells starting at address M_2, $M_2 \neq 0$. Next order taken sequentially.

PRT M_1M_2 (PRinT) $\Rightarrow$ Print M_1 words from consecutive memory cells starting at address M_2. Next order taken sequentially.

STP M_1M_2 (SToP) $\Rightarrow$ Stop computing. List M_1 and m_1 on console typewriter; when computing is started again, next command is taken from memory location M_2 if $M_2 \neq 0$.

In addition to the above list of orders the two-address computer may have an Initial Start Button which causes the first word of the input unit to be sent to the CR before the computer commences operation. This may be used to introduce a load order into the control register. The initial start button also causes the next order to be taken from memory location M_2 designated in the load command. Thus, if the second word in the input unit is not the first order of the routine, then it must be an order which will not affect the computation and must be one in which the order address M_2 is that of the first order of the routine, e.g., UCD 0 M_2. Finally, it is seen that if we wish to add two words x and y in the memory to produce a sum z in a desired location of the memory, the orders shown in Table 1-6-1 are necessary; in this table the locations of the orders are

TABLE 1-6-1

Memory location	Order symbol		
	I	M_1	M_2
L	CAD	X	...
$L+1$	HAU	Y	...
$L+2$	STU	Z	...

$L, L+1$, and $L+2$. For arithmetic orders, then, the two-address computer may require up to three orders for each order of a three-address computer. On the other hand, the ability to designate the next order to be performed allows the programmer more independence in coding sections of a problem. The flow diagram of Fig. 1-4-2 may again be used to code $x = \sqrt{y}$ using the orders of the two-address computer. Since this process is much the same as that for the three-address computer, it will not be given here.

1-7. Remarks

After discussing number bases and the scaling of numbers, we will consider again the requirements of flow diagrams. In such discussions we will introduce the *problem flow diagram* which describes the computational procedure for the problem and the *computer flow diagram* which describes the problem flow diagram in terms of the computer used. Finally, both flow diagrams will be replaced by a *detailed flow diagram* which will be tied to the problem in sufficient detail so that one may code from it, but it will not be so complete that it specifies the computer to be coded. Thus, the detailed flow diagram will allow us to carry through the major efforts of programming without referring directly to a particular computer.

PROBLEMS

1-1. Using the flow diagram of Fig. 1-4-1, code

$$s = \sum_{j=1}^{n} a_j$$

for the two-address computer.

1-2. Using the flow diagram of Fig. 1-4-2, code $x = \sqrt{y}$ for the two-address computer.

1-3. The answer $x = x_{i+1}$ for $x = \sqrt{y}$, $y > 0$ may be checked by showing that $|x_{i+1} \cdot x_{i+1} - y| \leq \epsilon$. Incorporate this check in the flow diagram of Fig. 1-4-2.

1-4. The arithmetic mean $\bar{x}$ is defined by

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

Assume a set of x_i, $1 \leq i \leq N$, to be given. Draw the flow diagram for the computation of the arithmetic mean.

1-5. Code the flow diagram of Prob. 1-4 for the three-address computer and for the two-address computer.

1-6. The sample variance s_x^2, is defined by

$$s_x^2 = \frac{\sum_{i=1}^{N} (x_i - \bar{x})^2}{N-1} = \frac{\sum_{i=1}^{N} x_i^2 - \frac{1}{N}\left(\sum_{i=1}^{N} x_i\right)^2}{N-1}$$

Incorporate the flow diagram of Prob. 1-4 in the flow diagram for the sample variance.

1-7. Code the flow diagram of Prob. 1-6 for the three-address computer and for the two-address computer.

2

NUMBER REPRESENTATIONS, ARITHMETIC OPERATIONS, AND SCALING

2-1. Introduction

In this chapter some basic concepts such as the representation of a number by a sequence of digits; the methods by which the computer performs the arithmetic operations of addition, subtraction, multiplication, and division; and the adaptation of data encountered in a problem for a computer are discussed. The representation of a number by a sequence of digits may be defined in terms of a power series, but a power series need not have a number representation. That there is not a one-to-one relation between number representation and power series leads to a discussion of how arithmetic operations are performed in a computer and how the numbers are stored in a computer. Knowing how the numbers are stored and operated upon by the computer leads to a discussion of how numerical data are scaled so that problems may be computed.

2-2. Number Representations

The number representations we discuss are constructed with the aid of several basic concepts. One of these is the definition of the *power* of a quantity as defined in college algebra texts. Another basic concept is that if we choose a positive integer a as a base or radix for a number representation then the digits of this representation are $a - 1$, $a - 2$, $a - 3$, . . . , $a - a = 0$. Thus, the ten digits 9, 8, 7, 6, 5, 4, 3, 2, 1, and 0 are the digits used for a base-10 representation of a number. The smallest base we consider is 2, and numbers represented with this base contain only the digits 1 and 0.

To represent a number in the base a, let the digit position be designated by k and the digit by α_k; i.e., the number

$$\alpha_{-n}\alpha_{-n+1} \cdots \alpha_{-1}\alpha_0.\alpha_1 \cdots \alpha_{m-1}\alpha_m \tag{2-2-1}$$

by definition represents the number

$$\alpha_{-n}a^n + \alpha_{-n+1}a^{n-1} + \cdots + \alpha_{-1}a^1 + \alpha_0a^0 + \alpha_1a^{-1} + \cdots + \alpha_{m-1}a^{-m+1} + \alpha_ma^{-m} \tag{2-2-2}$$

where, as above, each of the digits α_k satisfies the inequality $0 \leq \alpha_k \leq a - 1$. For example, the base-10 number representation 2,369.786 means

$$2 \times 10^3 + 3 \times 10^2 + 6 \times 10^1 + 9 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2} + 6 \times 10^{-3}$$

and a negative quantity of this magnitude is represented by $-2{,}369.786$ which means

$$-[2 \times 10^3 + 3 \times 10^2 + 6 \times 10^1 + 9 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2} + 6 \times 10^{-3}] = -2 \times 10^3 - 3 \times 10^2 - 6 \times 10^1 - 9 \times 10^0 - 7 \times 10^{-1} - 8 \times 10^{-2} - 6 \times 10^{-3}$$

The digits 10101.0 as the base-2 representation of a number mean

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1}$$

and, as the base-10 representation of a number, mean

$$1 \times 10^4 + 0 \times 10^3 + 1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0 + 0 \times 10^{-1}$$

The base-10 equivalent of the base-2 number 10101.0 is

$$1 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 + 0 \times 0.5 = 16 + 4 + 1 = 21.0$$

and we see that it is essential to know the base of the number representation in order that there will not be confusion as to its magnitude. It should be noted that the period (*decimal point* for base 10 and *binary point* for base 2 or *radix point* in general) is used to separate the integral part from the fractional part of the number representation.

We will discuss numbers represented in base 16, base 10, base 8, and base 2. Table 2-2-1 shows the equivalent number representations in bases 10, 8, and 2 of the sixteen base-16 digits. Here the six letters a through f have been used to represent the base-16 digits equivalent to the base-10 numbers 10 through 15 respectively.

TABLE 2-2-1. Digits of Base-16 Number Representations

Base-16 digits	Equivalent number representation		
	Base 10	Base 8	Base 2
0	00	00	0000
1	01	01	0001
2	02	02	0010
3	03	03	0011
4	04	04	0100
5	05	05	0101
6	06	06	0110
7	07	07	0111
8	08	10	1000
9	09	11	1001
a	10	12	1010
b	11	13	1011
c	12	14	1100
d	13	15	1101
e	14	16	1110
f	15	17	1111

2-3. Arithmetic Operations

In this section we give a simplified discussion of the arithmetic operations of addition, subtraction, multiplication, and division of two numbers. That is, we describe the rules of addition and then describe subtraction, multiplication, and division in terms of these rules. The discussion is carried out in this fashion because it reduces the number of rules that must be memorized and because it represents the actual operation of many of the high-speed arithmetic units.

It is common practice to teach the addition of two positive decimal digits in the early years of grammar school. This education usually begins with obtaining the sum of two digits by starting with one of the digits and counting up through the magnitude of the other. The result of repeated performances of problems of this type is the memorizing of the base-10 addition table. Addition tables for base-2 (binary), base-8 (octal), base-10 (decimal), and base-16 (hexadecimal or sexadecimal) digits are given in Tables 2-3-1, 2-3-2, 2-3-3, and 2-3-4. In each table the sum is obtained by entering the table with one digit as a row entry and the other digit as a column entry. The sum appears where that column and row cross. Thus, if a_{ij} is the element in both the ith row

and jth column of the base-a addition table, then

$$a_{ij} = a_{i0} + a_{0j} = a_{j0} + a_{0i} = a_{ji} = i + j$$

where $a_{i0} = i$ and $a_{0j} = j$, $0 \leq i, j \leq a - 1$. These tables may also be used for the subtraction of numbers contained in each table. For example, by entering the appropriate table with the subtrahend as a_{i0} (or a_{0j}) and going along that row (or column) until one reaches the a_{ij}, which equals the minuend, then the difference is a_{0j} (or a_{i0}). Thus, we are asking: what number (the difference) when added to the subtrahend equals the minuend?

TABLE 2-3-1. Binary Addition

	0	1
1	1	10

TABLE 2-3-2. Octal Addition

	0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	10	
2	3	4	5	6	7	10	11	
3	4	5	6	7	10	11	12	
4	5	6	7	10	11	12	13	
5	6	7	10	11	12	13	14	
6	7	10	11	12	13	14	15	
7	10	11	12	13	14	15	16	

TABLE 2-3-3. Decimal Addition

	0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10	
2	3	4	5	6	7	8	9	10	11	
3	4	5	6	7	8	9	10	11	12	
4	5	6	7	8	9	10	11	12	13	
5	6	7	8	9	10	11	12	13	14	
6	7	8	9	10	11	12	13	14	15	
7	8	9	10	11	12	13	14	15	16	
8	9	10	11	12	13	14	15	16	17	
9	10	11	12	13	14	15	16	17	18	

Now let us consider the addition of two positive numbers, each of which contains one or more digits. For example, the decimal addition

$$\begin{array}{r} 3{,}256.912 \\ +\ 9{,}563.827 \\ \hline 12{,}820.739 \end{array}$$

is learned as a serial addition of columns of digits starting with the rightmost column and proceeding leftwards column by column. Since the

Table 2-3-4. Hexadecimal Addition

0	1	2	3	4	5	6	7	8	9	*a*	*b*	*c*	*d*	*e*	*f*
1	2	3	4	5	6	7	8	9	*a*	*b*	*c*	*d*	*e*	*f*	10
2	3	4	5	6	7	8	9	*a*	*b*	*c*	*d*	*e*	*f*	10	11
3	4	5	6	7	8	9	*a*	*b*	*c*	*d*	*e*	*f*	10	11	12
4	5	6	7	8	9	*a*	*b*	*c*	*d*	*e*	*f*	10	11	12	13
5	6	7	8	9	*a*	*b*	*c*	*d*	*e*	*f*	10	11	12	13	14
6	7	8	9	*a*	*b*	*c*	*d*	*e*	*f*	10	11	12	13	14	15
7	8	9	*a*	*b*	*c*	*d*	*e*	*f*	10	11	12	13	14	15	16
8	9	*a*	*b*	*c*	*d*	*e*	*f*	10	11	12	13	14	15	16	17
9	*a*	*b*	*c*	*d*	*e*	*f*	10	11	12	13	14	15	16	17	18
a	*b*	*c*	*d*	*e*	*f*	10	11	12	13	14	15	16	17	18	19
b	*c*	*d*	*e*	*f*	10	11	12	13	14	15	16	17	18	19	1*a*
c	*d*	*e*	*f*	10	11	12	13	14	15	16	17	18	19	1*a*	1*b*
d	*e*	*f*	10	11	12	13	14	15	16	17	18	19	1*a*	1*b*	1*c*
e	*f*	10	11	12	13	14	15	16	17	18	19	1*a*	1*b*	1*c*	1*d*
f	10	11	12	13	14	15	16	17	18	19	1*a*	1*b*	1*c*	1*d*	1*e*

augend, 3,256.912, represents the series

$$3 \times 10^3 + 2 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 + 9 \times 10^{-1} + 1 \times 10^{-2} + 2 \times 10^{-3}$$

and the addend 9,563.827 represents the series

$$9 \times 10^3 + 5 \times 10^2 + 6 \times 10^1 + 3 \times 10^0 + 8 \times 10^{-1} + 2 \times 10^{-2} + 7 \times 10^{-3}$$

the sum of these two series is

$$12 \times 10^3 + 7 \times 10^2 + 11 \times 10^1 + 9 \times 10^0 + 17 \times 10^{-1} + 3 \times 10^{-2} + 9 \times 10^{-3} \quad (2\text{-}3\text{-}1)$$

and the coefficients of the series do not form a legitimate number representation since the coefficients of 10^{-1}, 10^{1}, and 10^{3} are themselves numbers exceeding one-digit representations. However, the sum of the two numbers [series (2-3-1)] may be changed to a power series, the coefficients of which form a legitimate number representation. (Recall that a number representation may always be interpreted as a power series, but that the converse may not be true.) Expression (2-3-1) can be changed to a power series, the coefficients of which have a number representation with the aid of a simple rule. The rule is: Start with the rightmost coefficient of the power of 10 which has a 2-digit representation. Replace the coefficient by its second digit and add 1 to the coefficient of the next higher power of 10. Repeat this process until all two-digit coefficients have been removed from the power series.

For example, when the above rule is applied to expression (2-3-1) the result is

$$1 \times 10^{4} + 2 \times 10^{3} + 8 \times 10^{2} + 2 \times 10^{1} + 0 \times 10^{0} + 7 \times 10^{-1} + 3 \times 10^{-2} + 9 \times 10^{-3}$$

which has the base-10 number representation 12,820.739.

To state algebraically the addition of two positive numbers of the same base a, let

$$\alpha = \sum_{r=-n}^{m} \alpha_r a^{-r} = \alpha_{-n}a^{n} + \cdots + \alpha_{-1}a^{1} + \alpha_0 a^{0} + \alpha_1 a^{-1} + \cdots + \alpha_m a^{-m}$$

and

$$\alpha' = \sum_{r=-n}^{m} \alpha'_r a^{-r} = \alpha'_{-n}a^{n} + \cdots + \alpha'_{-1}a^{1} + \alpha'_0 a^{0} + \alpha'_1 a^{-1} + \cdots + \alpha'_m a^{-m}$$

be the two positive numbers where zero coefficients have been inserted where necessary to make the two series contain the same powers of a. Let α'' be the sum of α and α', then

$$\alpha + \alpha' = \sum_{r=-n}^{m} (\alpha_r + \alpha'_r)a^{-r} = \alpha'' = c_{-n}a^{n+1} + \sum_{r=-n}^{m} \alpha''_r a^{-r} \qquad (2\text{-}3\text{-}2)$$

where $c_{m+1} = 0$ and

$$\alpha''_r = \alpha_r + \alpha'_r + c_{r+1} \quad \text{and} \quad c_r = 0 \quad \text{if } \alpha_r + \alpha'_r + c_{r+1} < a \qquad (2\text{-}3\text{-}3)$$

$$\alpha''_r = \alpha_r + \alpha'_r + c_{r+1} - a \quad \text{and} \quad c_r = 1 \quad \text{if } \alpha_r + \alpha'_r + c_{r+1} \geq a \qquad (2\text{-}3\text{-}4)$$

and c_r is the carry digit from the coefficient of a^{-r} to the coefficient of a^{-r+1}. That c_r, the *carry digit*, will never be greater than 1 may be seen by examining the largest possible sum in (2-3-4) starting with $r = m$ where $c_{m+1} = 0$ and continuing the examination with the other coefficients.

In learning how to subtract one positive number from another, a child usually starts with problems in which the minuend and subtrahend are positive and the minuend is greater than the subtrahend. Eventually, though, he is faced with problems in which the subtrahend is greater than the minuend. At first for these problems he is usually taught to start with the minuend and count down through the subtrahend reaching a negative number, which is the difference. Later he is taught that if he subtracts the minuend from the subtrahend and attaches a negative sign to the difference, the answer thus obtained is the same as that by the previous method. Finally, he is taught the addition of two numbers of different signs and is shown the equivalence between subtraction and changing the sign of the subtrahend and adding. In designing a high-speed arithmetic unit one soon realizes that the first rule will involve a long counting procedure when the magnitude of the subtrahend is large. The second rule for subtraction involves a circular argument; that is, one must know how to compare numbers before he can subtract. Although the human being can perform these comparisons visually and make rapid decisions, he is still performing a mental subtraction and it is this subtraction which must be incorporated in the design of the computer before it can be designed to subtract the subtrahend from the minuend. Application of the third rule implies that the addition of two numbers with different signs can be performed without subtraction. We next describe one of several methods by which such an addition can be performed.

To complete our discussion of the addition of two numbers, we find there are three cases to be considered. The first, which has already been discussed, is the addition of two positive numbers. The second is the addition of two negative numbers; and the third is the addition of two numbers, one of which is positive and the other is negative. The addition of two negative numbers is performed by adding their magnitudes and attaching a minus sign to the sum; i.e.,

$$-\alpha + (-\alpha') = -(\alpha + \alpha') = -\alpha''$$

Thus, our three cases are substantially only two; namely, the adding of two numbers with like signs and the adding of two numbers with unlike signs. The determination of whether the two numbers have like or unlike signs is a simply automated comparison; however, the comparison of two

magnitudes is much more complex. This comparison of the magnitude of two numbers is avoided by the use of a mathematical procedure called *complementation.*

The computer addition of two numbers of unlike sign depends upon the ability to form from α_r, when desired, a digit β_r such that

$$\beta_r = a - \alpha_r - 1 = (a - 1) - \alpha_r \tag{2-3-5}$$

The digit β_r is defined as the $(a - 1)$ complement of α_r. In some computers the complement is obtained by reversing the mode of digit examination. The binary computer has the simplest complementing system since the complement of 1 is 0 and vice versa. Now, let us consider the sum of $\alpha + (-\alpha')$ or $(-\alpha') + \alpha$ where as before

$$\alpha = \sum_{r=-n}^{m} \alpha_r a^{-r} \tag{2-3-6}$$

and

$$\alpha' = \sum_{r=-n}^{m} \alpha'_r a^{-r} \tag{2-3-7}$$

(If the indices of the summations appearing in Eqs. (2-3-6) and (2-3-7) are not the same, they are made so by inserting zero coefficients in the leading end or the trailing end or both of one or both numbers.) If the sum $\alpha + (-\alpha')$ is written as

$$\alpha + (a^{n+1} - \alpha') - a^{n+1} \tag{2-3-8}$$

and if we write

$$a^{n+1} = a^{-m} + \sum_{r=-n}^{m} (a - 1)a^{-r} = (a - 1)a^n + \cdots + (a - 1)a^1 + (a - 1)a^0 + (a - 1)a^{-1} + \cdots + (a - 1)a^{-m} + a^{-m} \tag{2-3-9}$$

then from Eqs. (2-3-5) and (2-3-9) the quantity in parenthesis of the expression (2-3-8) becomes

$$a^{n+1} - \alpha' = a^{-m} + \sum_{r=-n}^{m} (a - 1)a^{-r} - \sum_{r=-n}^{m} \alpha'_r a^{-r}$$

$$= a^{-m} + \sum_{r=-n}^{m} [(a - 1) - \alpha'_r]a^{-r} = a^{-m} + \sum_{r=-n}^{m} \beta'_r a^{-r} = a^{-m} + \beta'$$

where β' is a number formed by replacing each digit of α' by its $(a - 1)$ complement. The number $\bar{\alpha}'$ is formed by adding a^{-m} to β'

$$\bar{\alpha}' = \beta' + a^{-m} = \sum_{r=-(n+1)}^{m} \bar{\alpha}'_r a^{-r}$$

and is called the *complement* of α'. Here $\bar{\alpha}'_{-(n+1)} = 0$ except when $\alpha' = 0$ and then $\bar{\alpha}'_{-(n+1)} = 1$ and all other $\bar{\alpha}'_r = 0$. Note that $\bar{\alpha}'$ is a positive number formed from the addition of two positive numbers β' and a^{-m}. If we let $\alpha'' = \alpha + \bar{\alpha}'$, then the quantity $\alpha + (a^{n+1} - \alpha')$ of the expression (2-3-8) may be obtained by addition as in Eqs. (2-3-2) to (2-3-4) and the expression (2-3-8) becomes

$$\alpha + (a^{n+1} - \alpha') - a^{n+1} = \alpha + \bar{\alpha}' - a^{n+1} = \alpha'' - a^{n+1}$$
$$= c_{-n}a^{n+1} + \sum_{r=-n}^{m} \alpha''_r a^{-r} - a^{n+1}$$

where c_{-n} can be only 1 or 0. If $c_{-n} = 1$, then

$$\alpha + (a^{n+1} - \alpha') - a^{n+1} = \sum_{r=-n}^{m} \alpha''_r a^{-r}$$

and in this case $\alpha - \alpha'$ is a positive number. If $c_{-n} = 0$, then

$$\alpha + (a^{n+1} - \alpha') - a^{n+1} = -a^{n+1} + \sum_{r=-n}^{m} \alpha''_r a^{-r} = -\left(a^{n+1} - \sum_{r=-n}^{m} \alpha''_r a^{-r}\right)$$
$$= -\sum_{r=-n}^{m} \bar{\alpha}''_r a^{-r} = -\bar{\alpha}''$$

$\bar{\alpha}''$ is the complement of the positive number $\alpha + (a^{n+1} - \alpha')$, and the coefficient $\bar{\alpha}''_{-(n+1)} = 0$. In this case $\alpha - \alpha'$ is a negative number. Thus, the sum of two numbers of unlike signs may be obtained by complementing the negative number and forming the sum of two positive numbers: the original positive number and the complemented number. If in this sum the carry digit $c_{-n} = 1$, the result is taken to be the positive number obtained by dropping this carry digit; and if $c_{-n} = 0$, the result is obtained by complementing the sum and attaching thereto a minus sign.

For examples of the addition of two numbers of unlike sign, consider first the addition of the two base-10 numbers 3,963.200 + (−0,250.735). The complement of 0,250.735 is

$$9{,}749.265 = 9{,}749.264 + 0.001 = 10^5 - 0{,}250.735$$

and

$$\begin{array}{r} 3{,}963.200 \\ +9{,}749.265 \\ \hline \textcircled{1}3{,}712.465 \end{array}$$

In this result $c_{-3} = 1$, circled, and the result is

$$3{,}712.465 = 3{,}963.200 + (-0250.735)$$

obtained by dropping c_{-3}. As a second example, consider 0,250.735 + (−3,963.200). The complement of 3,963.200 is

$$6{,}036.800 = 6{,}036.799 + 0.001 = 10^5 - 3{,}963.200$$

and

$$\begin{array}{r} 0{,}250.735 \\ +6{,}036.800 \\ \hline \textcircled{0}6{,}287.535 \end{array}$$

In this result $c_{-3} = 0$, circled, and the answer is obtained by complementing this result and indicating that the result is negative. The complement of 6,287.535 is 3,712.465 and the result is −3,712.465.

As a third example, consider an addition involving two binary numbers which are the equivalent of the base-10 numbers 25.25 and 10.5. Since $25.25 = 16 + 8 + 1 + \frac{1}{4} = 2^4 + 2^3 + 2^0 + 2^{-2}$, then the binary representation of 25.25 is 11001.01. Similarly,

$$10.5 = 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2}$$

and its binary representation is 01010.10. To form the sum 11001.01 + (−01010.10), form the complement of 01010.10, which is

$$10101.10 = 10101.01 + .01 = 2^5 - 01010.10$$

and then

$$\begin{array}{r} 11001.01 \\ +10101.10 \\ \hline \textcircled{1}01110.11 \end{array}$$

Since $c_{-4} = 1$, circled, then the answer is 1110.11 obtained by dropping c_{-4} in the above result. The base-10 equivalent of this result is, of course, 14.75. If, however, one forms the sum 01010.10 + (−11001.01), the complement of 11001.01 is

$$00110.11 = 00110.10 + .01 = 2^5 - 11001.01$$

and

$$\begin{array}{r} 01010.10 \\ +00110.11 \\ \hline \textcircled{0}10001.01 \end{array}$$

Since $c_{-4} = 0$, circled, then the answer desired is obtained by complementing the above result and indicating that the answer is negative. Thus the answer is $-1110.11 = -(01110.10 + .01)$.

For many computers with double-length accumulators it is possible to add a number to either the most significant or the least significant half

of the accumulator. It is essential that the programmer recognize that adding to either half of the accumulator may affect the other half. This is illustrated in the following example. Consider the addition of -444 to the upper half A_U of a 6-digit double-length accumulator which contains + 333 333. The complement of -444 is + 556. The addition is

A_U	A_L
+333	333
+556	
⓪889	333

and, since the highest-order carry is zero (circled) after adding the complement, the sum must be uncomplemented and indicating that the result is negative. The accumulator then contains

A_U	A_L
−110	667

The contents of the lower half A_L of the accumulator has changed to the complement 667 of its original contents 333.

The subtraction of one number from another, whether the two numbers have the same sign or have different signs, is defined in terms of addition: *To subtract the subtrahend from the minuend, change the sign of the subtrahend and add it to the minuend.* Thus, subtraction can be obtained by a simple modification of the addition process. Figure 2-3-1 shows a flow diagram for the addition, $\alpha + \alpha' = \alpha''$, or subtraction, $\alpha - \alpha' = \alpha''$, of two numbers. The notation for this figure is that $\alpha = s|\alpha|$ and $\alpha' = s'|\alpha'|$ where $s = 1$ if $\alpha \geq 0$ and $s = -1$ if $\alpha < 0$, and similarly $s' = 1$ if $\alpha' \geq 0$ and $s' = -1$ if $\alpha' < 0$. Barred quantities are used to represent the complement.

Most of the existing digital computers use the sign and magnitude system described above and a process similar to that shown in Fig. 2-3-1 to implement the addition and subtraction orders. In some computers negative numbers are stored in the complement form just described. In these computers the complementing operation indicated by the barred quantities in boxes 4, 6, and 9 of Fig. 2-3-1 is not required in the process for computing the sum since negative quantities are left in the complement form selected for the computer. It does not take long for a programmer or computer user to become adept at recognizing, or obtaining by conversion, the magnitude of a number in complement form. Desk calculator operators, for example, frequently can convert a result in complement form as fast as they can write. It is customary, however, to convert all final results to sign and magnitude form.

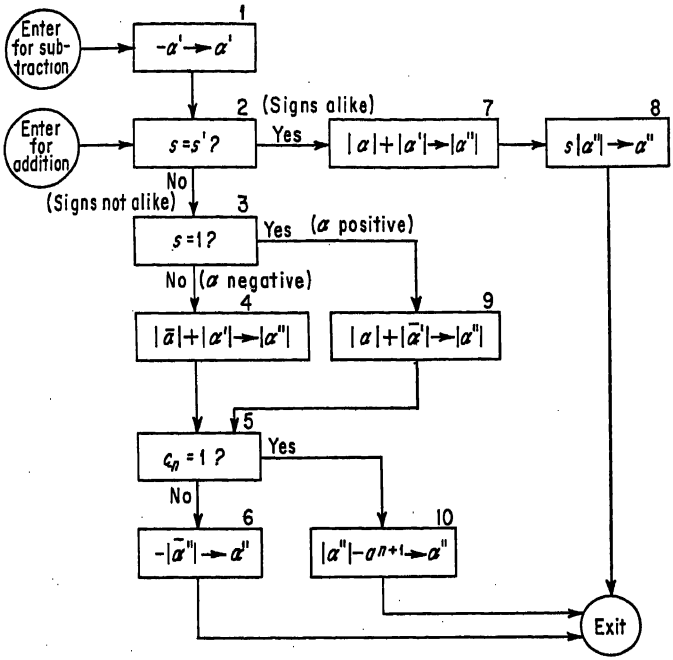


FIG. 2-3-1. Flow diagram for computer addition and subtraction of two numbers. (A barred symbol designates the complement of the corresponding number.)

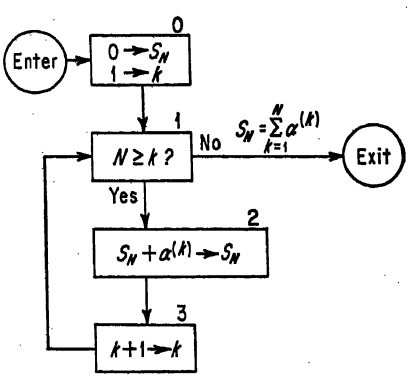


FIG. 2-3-2. Flow diagram for $S_N = \sum_{k=1}^{N} \alpha^{(k)}$.

Before considering the multiplication of two numbers, represented in the sign and magnitude system, let us consider one of the subprocesses involved in the description of multiplication. A generalization of this subprocess is that of obtaining the sum of a finite series of positive numbers which is basically the problem of Eq. (1-4-1). The problem is restated here with different symbols. Let $S_N = \sum_{k=1}^{N} \alpha^{(k)}$ where the superscript (k) is used to differentiate between the N numbers $\alpha^{(1)}$, $\alpha^{(2)}$, . . . , $\alpha^{(N)}$. The superscript is enclosed in parentheses to distinguish it from a power. A subscript will be used to indicate the digits in the number $\alpha^{(k)}$. Considering the formation of S_N as a computer problem, the flow diagram for the computation of S_N is given in Fig. 2-3-2. The product of a positive number α by a digit N, $1 \leq N \leq a - 1$, is defined as

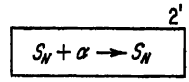


FIG. 2-3-3. Change to Fig. 2-3-2 producing $S_N = N\alpha$.

$$N\alpha = S_N = \sum_{k=1}^{N} \alpha^{(k)}$$

where all $\alpha^{(k)} = \alpha$, and as $N\alpha = 0$ if $N = 0$. Thus, the product $N\alpha$ may be computed by the routine given in the flow diagram of Fig. 2-3-2 by replacing box 2 with box 2' of Fig. 2-3-3. It should be noted in Fig. 2-3-2 that by placing the test, $N \geq k$, before the summing operation, $S_N + \alpha \rightarrow S_N$, the product $N\alpha = 0$ for $N = 0$ is also obtained.

The product of two numbers α and α' is positive if α and α' have the same sign and is negative if they have different signs. By comparing the signs of the multiplier and multiplicand, the sign of the product may be determined. The product can be computed by forming the product of the absolute values of the multiplicand and the multiplier and attaching the proper sign to the result. Our discussion is thus confined to the product of two positive numbers. Again, let α and α' be defined by Eqs. (2-3-6) and (2-3-7) respectively, then

$$\alpha\alpha' = \alpha \sum_{r=-n}^{m} \alpha'_r a^{-r} = \sum_{r=-n}^{m} \alpha'_r (a^{-r}\alpha) \tag{2-3-10}$$
$$= \alpha'_{-n}(a^n\alpha) + \alpha'_{-n+1}(a^{n-1}\alpha) + \cdots + \alpha'_m(a^{-m}\alpha)$$

By defining

$$\alpha^{(-r+1)} = a\alpha^{(-r)} = a^{-r+1}\alpha \tag{2-3-11}$$

then Eq. (2-3-10) may be rewritten as

$$\alpha\alpha' = \sum_{r=-n}^{m} \alpha'_r \alpha^{(-r)} \tag{2-3-12}$$

Let $p = \alpha\alpha'$, then p may be computed by first forming $\alpha^{(-m)} = \alpha^{-m}\alpha$ by shifting α to the right m digits with respect to the radix point. Next, set $p = 0$ and then form $\alpha'_m\alpha^{(-m)}$ by adding $\alpha^{(-m)}$ to p, α'_m times. Then shift $\alpha^{(-m)}$ one digit to the left, producing $a\alpha^{(-m)} = \alpha^{(-m+1)}$, and add $\alpha^{(-m+1)}$ to p, α'_{m-1} times. Continuing in this fashion, the last step is to add $\alpha^{(n)}$ to p, α'_{-n} times, after which $p = \alpha\alpha'$. The flow diagram for the multiplication of two positive numbers by repeated additions and shifts is given in Fig. 2-3-4.

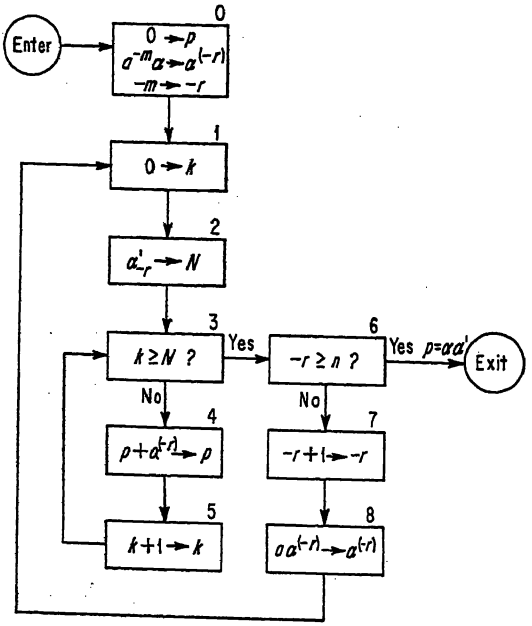


FIG. 2-3-4. Flow diagram for $p = \alpha\alpha'$.

For this flow diagram, it is assumed that the coefficients α'_r of the number α' are themselves stored as digits in the arithmetic unit and that no information is lost by shifting α m digits to the right. In most computers the arithmetic unit has a double-length accumulator for this purpose, and multiplication is implemented by an automatic operation executed by a process such as that shown in Fig. 2-3-4.

To program the routine given in the flow diagram of Fig. 2-3-4 by the three-address computer described in Chap. 1, it is necessary to add to that computer the following commands:

SHL $M_1M_2M_3 \Rightarrow m_1 \times a^{M_2} = m_3$ where a is the base of the number representation used in the computer; i.e., shift each digit of the word stored in M_1 to the left M_2 digit positions and store the result in memory location M_3.

SHR $M_1M_2M_3 \Rightarrow m_1 \times a^{-M_2} = m_3$ where a is the base of the number representation used in the computer; i.e., shift each digit of the word stored in M_1 to the right M_2 digit positions and store the result in memory location M_3.

(As new orders are needed to improve the operation of our computers, they will be added in the text. A complete set of orders is given in Appendix I for the three-address computer and in Appendix II for the two-address computer. Often we will introduce only the three-address orders in the text since an adequate description of the corresponding two-address order is given in Appendix II. The reader may find it helpful in using the appendices to place a check mark next to the orders in the appendices as he learns about them in the text.)

As an example of multiplication by the flow diagram of Fig. 2-3-4, consider the product of the two base-10 numbers 12.6 and 10.5. Let

$$\alpha = 12.6 = 1 \times 10^1 + 2 \times 10^0 + 6 \times 10^{-1}$$

and

$$\alpha' = 10.5 = 1 \times 10^1 + 0 \times 10^0 + 5 \times 10^{-1}$$

then for Eq. (2-3-7) $a = 10$, $n = 1$, $m = 1$, $\alpha'_1 = 5$, $\alpha'_0 = 0$, $\alpha'_{-1} = 1$, and

$$\alpha^{(-m)} = a^{-1}\alpha = 10^{-1}(1 \times 10^1 + 2 \times 10^0 + 6 \times 10^{-1})$$
$$= 1 \times 10^0 + 2 \times 10^{-1} + 6 \times 10^{-2} = 1.26$$

The progress of the computation of $p = \alpha\alpha'$ by the flow diagram is shown in Table 2-3-5. A table such as Table 2-3-5 is often referred to as a

TABLE 2-3-5. Partial Dynamic Storage Chart for Multiplication

n	$-r$	$N = \alpha'_r$	k	$\alpha^{(-r)}$	p
1	−1	5	0	1.26	0000.00
1	−1	5	1	1.26	0001.26
1	−1	5	2	1.26	0002.52
1	−1	5	3	1.26	0003.78
1	−1	5	4	1.26	0005.04
1	−1	5	5	1.26	0006.30
1	0	0	0	12.60	0006.30
1	1	1	0	126.00	0006.30
1	1	1	1	126.00	0132.30

partial storage chart because it shows the contents of a part of the computer. This particular storage chart is referred to as a *partial dynamic storage chart* because it shows how the storage changes each time a specified set of operations is performed by the computer. The rows in the table show the state of the calculation on each pass through box 3 of Fig. 2-3-4. The final answer is underlined.

The computer division procedure determines the structure of the arithmetic unit to a greater extent than the other arithmetic operations. Since the quotient is positive whenever the dividend and the divisor have the same sign and negative whenever they have different signs, the sign of the quotient may be determined by comparing the signs of the dividend and divisor. A discussion of the quotient of two positive numbers will suffice. Again, assume α and α' are two positive numbers, with $\alpha' \neq 0$, defined by Eqs. (2-3-6) and (2-3-7) respectively. Furthermore, we assume $\alpha'_{-n} > 0$. If this were not the case, the digits of the divisor could be shifted left until the condition was satisfied. This is always possible since α' was assumed to be greater than zero. Let it be desired to find α/α' for those positive α and α' such that $0 \leq \alpha/\alpha' < 1$. We obtain, instead, an approximate quotient

$$q = \sum_{s=1}^{k} q_s a^{-s} \doteq \frac{\alpha}{\alpha'} \tag{2-3-13}$$

where each of the q_s is one of the digits $0, 1, 2, \ldots, a - 1$. For a given number base, the digital representation of the exact quotient may have an infinite sequence of digits. Let

$$r^{(s)} = ar^{(s-1)} - q_s\alpha' \text{ or } q_s a^{-1} = \frac{-a^{-1}r^{(s)}}{\alpha'} + \frac{r^{(s-1)}}{\alpha'} \tag{2-3-14}$$

where $r^{(0)} = \alpha$ and $0 \leq r^{(s)} < \alpha'$;

then

$$\begin{aligned}
q_1 a^{-1} &= \frac{-a^{-1}r^{(1)}}{\alpha'} + \frac{\alpha}{\alpha'} \\
q_2 a^{-2} &= \frac{-a^{-2}r^{(2)}}{\alpha'} + \frac{a^{-1}r^{(1)}}{\alpha'} \\
q_3 a^{-3} &= \frac{-a^{-3}r^{(3)}}{\alpha'} + \frac{a^{-2}r^{(2)}}{\alpha'} \\
&\cdots\cdots\cdots\cdots\cdots\cdots \\
q_k a^{-k} &= \frac{-a^{-k}r^{(k)}}{\alpha'} + \frac{a^{-k+1}r^{(k-1)}}{\alpha'}
\end{aligned}$$

and by adding the above

$$q = \sum_{s=1}^{k} q_s \alpha^{-s} = \frac{\alpha}{\alpha'} - \frac{\alpha^{-k} r^{(k)}}{\alpha'} \tag{2-3-15}$$

From Eq. (2-3-15) it is seen that the difference $\alpha/\alpha' - q$ can be made as small as desired by choosing k sufficiently large, since $r^{(k)}/\alpha' < 1$. The $r^{(s)}$ are called the *remainders* and the restriction that $0 \leq r^{(s)} < \alpha'$ indicates how the computer is to carry out the division. The procedure outlined is called the Euclidean algorithm. Start by subtracting α' from $a\alpha$, and if $a\alpha - \alpha'$ is positive, then α' is subtracted from $a\alpha - \alpha'$. If $a\alpha - 2\alpha'$ is positive, then α' is subtracted from $a\alpha - 2\alpha'$, etc. Finally, $a\alpha - (p+1)\alpha'$ will be negative and $a\alpha - p\alpha'$ will be nonnegative for some p, $p = 0, 1, 2, \ldots, a-1$; then $q_1 = p$ and $r^{(1)} = a\alpha - q_1\alpha'$. This process is now repeated with α replaced by $ar^{(1)}$, and q_2 is similarly obtained. This procedure is continued until q_k is obtained. Figure 2-3-5 shows the flow diagram for the division of two positive numbers where the only arithmetic operations used are subtraction, addition, and shifting. In this flow diagram Eq. (2-3-14) is considered to be the principal equation to be computed. In box 0, s is set equal to 1 and $ar^{(0)}$ is set equal to $a\alpha$. The operation of box 6 is that of shifting $r^{(s)}$ one digit position to the left. As an example of division, following the flow diagram of Fig. 2-3-5, consider the quotient of the two base-10

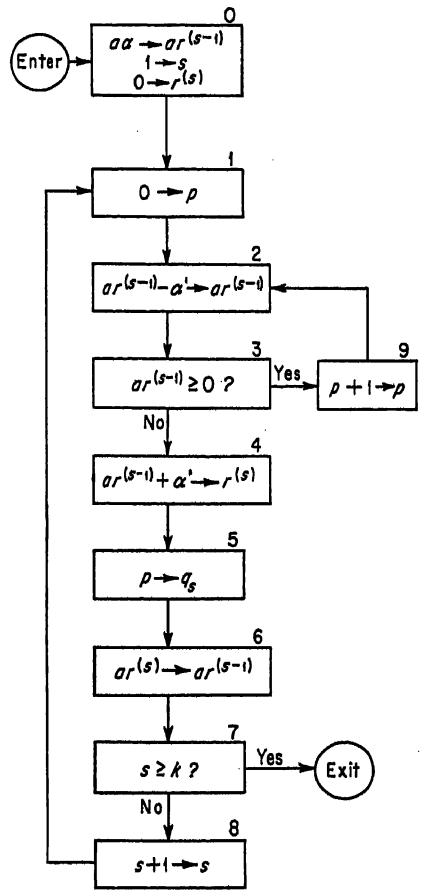


FIG. 2-3-5. Flow diagram for the division of two positive numbers.

numbers 0.359/1.25. Letting $k = 3$, the division is shown in Table 2-3-6 and gives

$$q = \sum_{s=1}^{k=3} q_s a^{-s} = 2 \times 10^{-1} + 8 \times 10^{-2} + 7 \times 10^{-3} = .287$$

The rows in Table 2-3-6 show the state of the calculation after execution of the operations of the box indicated in column 1.

TABLE 2-3-6. Partial Dynamic Storage Chart for Division

Box	k	s	p	α'	$ar^{(s-1)}$	$r^{(s)}$	q_s
0	3	1		1.25	3.59	0.00	
3	3	1	0	1.25	2.34	0.00	
3	3	1	1	1.25	1.09	0.00	
3	3	1	2	1.25	−0.16	0.00	
5	3	1	2	1.25	−0.16	1.09	2
7	3	1	2	1.25	10.90	1.09	
3	3	2	0	1.25	9.65	1.09	
3	3	2	1	1.25	8.40	1.09	
3	3	2	2	1.25	7.15	1.09	
3	3	2	3	1.25	5.90	1.09	
3	3	2	4	1.25	4.65	1.09	
3	3	2	5	1.25	3.40	1.09	
3	3	2	6	1.25	2.15	1.09	
3	3	2	7	1.25	0.90	1.09	
3	3	2	8	1.25	−0.35	1.09	
5	3	2	8	1.25	−0.35	0.90	8
7	3	2	8	1.25	9.00	0.90	
3	3	3	0	1.25	7.75	0.90	
3	3	3	1	1.25	6.50	0.90	
3	3	3	2	1.25	5.25	0.90	
3	3	3	3	1.25	4.00	0.90	
3	3	3	4	1.25	2.75	0.90	
3	3	3	5	1.25	1.50	0.90	
3	3	3	6	1.25	0.25	0.90	
3	3	3	7	1.25	−1.00	0.90	
5	3	3	7	1.25	−1.00	0.25	7
7	3	3	7	1.25	2.50	0.25	

2-4. Scaling of Numbers

This subject will be introduced in the reverse order of its historical introduction to high-speed computers. The "scientific representation"

or "scientific notation" for numbers has long been used by chemists, physicists, engineers, and others who work with numbers which either vary over a wide range or numbers which are extremely large or extremely small when representing a measure in a desired unit. For example, an astronomer may be giving the distance, in light-years, to a distant star. If this distance is approximately 1,860,000,000 light-years and this is as accurate as it is measured (that is, the digits 1 and 8 are correct and the remaining digits are nearer 60,000,000 than they are 50,000,000 or 70,000,000), it may be preferable to write this distance as 1.86×10^9 light-years to conserve effort, space, and ambiguity. Similarly, the physicists may give a radius of an atom as 1.5×10^{-9} centimeters rather than fill the line with zeros.

A number α, the base of which is a, having $m + 1$ digits has the scientific representation

$$\alpha = (\pm \alpha_0.\alpha_1\alpha_2 \cdots \alpha_m) \times a^n = \pm a^n \sum_{r=0}^{m} \alpha_r a^{-r}$$

where $\alpha_0 > 0$; that is, $1 \leq \alpha_0.\alpha_1\alpha_2 \cdots \alpha_m < a$, and the exponent n is either a positive or negative integer or is zero. Usually, in this notation, $(m + 1)$ is the number of significant digits in α. The restriction $\alpha_0 > 0$ does not allow zero to be written in this form, and common usage has been to write $\alpha = 0$ for zero.

Let us now adapt the scientific notation to a computer word containing ten *decimal* digits. One representation of such a word is

$$\pm \alpha_0\alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6\alpha_7\alpha_8\alpha_9$$

where the digits α_r, $0 \leq r \leq 7$, are referred to as the *precision digits*, and the *power* or *exponent* is represented by the digits α_8 and α_9. Since the computer word has only one sign digit and since scientific notation requires a sign for the precision digits and a sign for the exponent, $a^2/2 = 50$ (for base 10) is usually added to the exponent. This restricts the range for the exponent to $-50 \leq n \leq 49$ (i.e., $-a^2/2 \leq n \leq a^2/2 - 1$) and then the modified or *pseudo exponent* $\bar{n} = n + 50$ (i.e., $n + a^2/2$) has the range $0 \leq \bar{n} \leq 99$ (i.e., $0 \leq \bar{n} < a^2 - 1$). Thus

$$\alpha = \pm \alpha_0\alpha_1 \cdots \alpha_7\alpha_8\alpha_9 = \pm a^{\alpha_8 a + \alpha_9 - a^2/2} \sum_{r=0}^{7} \alpha_r a^{-r} \qquad (2\text{-}4\text{-}1)$$

where $\bar{n} = \alpha_8\alpha_9$. A computer word having the representation of Eq. (2-4-1) is said to be represented in *floating point* notation as contrasted to the fixed point notation discussed later. A second form for representing floating point numbers using the first two digits for the pseudo

exponent is described at the end of this chapter. This second form has a computational advantage over the form described above. The absolute value of the largest number represented in this system is

$$9.9999999 \times 10^{49}$$

and the absolute value of the smallest nonzero number is

$$1.0000000 \times 10^{-50}$$

Zero is represented in this system by $\bar{n} = \alpha_8\alpha_9 = 00$ and

$$\alpha_0 = \alpha_1 = \cdots = \alpha_7 = 0$$

and operations involving zero are given special handling by the computer.

So that floating point calculations may be conveniently programmed for the three-address computer of Chap. 1, the following *extract command* is added to that computer.

XTR $M_1M_2M_3 \Rightarrow$ Replace the digits of m_3, as designated by m_2, with the corresponding digits of m_1. If in m_2, s or d_r ($r = 0, 1, \ldots, 9$) is a zero, the corresponding sign or digit of m_3 is not changed; and if s or d_r is a one, the corresponding sign or digit of m_3 is replaced by the corresponding sign or digit of m_1.

The extract command is also useful in many other computational operations.

As an example of a floating point operation, consider the product $\alpha'' = \alpha\alpha'$ where $\alpha > 0$ and $\alpha' > 0$. Writing $\bar{\alpha} = \alpha_0.\alpha_1 \cdots \alpha_7$, $\bar{n} = \alpha_8\alpha_9$, $\bar{\alpha}' = \alpha'_0.\alpha'_1 \cdots \alpha'_7$, and $\bar{n}' = \alpha'_8\alpha'_9$, then

$$\alpha = \bar{\alpha} \times 10^{\bar{n}-50}$$
$$\alpha' = \bar{\alpha}' \times 10^{\bar{n}'-50}$$

It is desired to form $\alpha'' = \bar{\alpha}'' \times 10^{\bar{n}''-50}$; where $\bar{\alpha}'' = \alpha''_0.\alpha''_1 \cdots \alpha''_7$, $1 \leq \bar{\alpha}'' < a$, and $\bar{n}'' = \alpha''_8\alpha''_9$, such that $\alpha'' \doteq \alpha\alpha'$ (the approximation symbol is used to indicate that the product is rounded and thus may not be exact). Let

$$\beta = \beta_0.\beta_1 \cdots \beta_7\beta_8\beta_9 \doteq a^{-1}\bar{\alpha}\bar{\alpha}'$$

where β_0 through β_9 are the leading 10 digits of the product $a^{-1}\bar{\alpha}\bar{\alpha}'$, then

$a^{-1} \leq \beta < a$. Set

$$\bar{\alpha}'' = \begin{cases} \beta_0.\beta_1 \cdots \beta_7 & \text{if } \beta_0 \neq 0 \\ \beta_1.\beta_2 \cdots \beta_8 & \text{if } \beta_0 = 0 \end{cases}$$

$$\bar{n}'' = \begin{cases} \bar{n} + \bar{n}' - 49 & \text{if } \beta_0 \neq 0 \\ \bar{n} + \bar{n}' - 50 & \text{if } \beta_0 = 0 \end{cases}$$

then α'' has the floating point representation, Eq. (2-4-1), and is $\alpha'' = \bar{\alpha}'' \times 10^{\bar{n}''-50}$ provided $0 \leq \bar{n} < 100$. If $\bar{n} \geq 100$, the case is called *exponent overflow*, and the computer is usually programmed to set an "alarm"; if $\bar{n} < 0$, the floating point computer product is made zero.

Let w = 01000000000 (sign not extracted), x = 11111111100 (sign extracted), and y = 00000000011 (sign not extracted) be the extractors to be used in the product computation; and let the symbol XTR $(m_1,m_2) \rightarrow m_3$ represent the extract command. The flow diagram for the preceding floating point product is given in Fig. 2-4-1.

In the flow diagram of Fig. 2-4-1 we have assumed that all words are

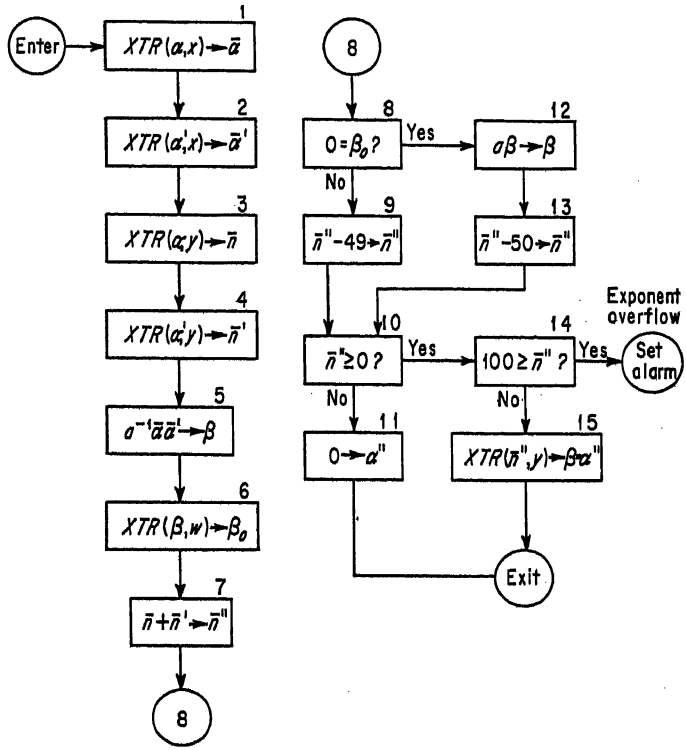


FIG. 2-4-1. Flow diagram for floating point multiplication.

stored in the computer in the form $\pm\alpha_0\alpha_1 \cdots \alpha_9$ without considering the construction of the arithmetic unit. This flow diagram may be changed by such considerations. Boxes 10 and 14 are safeguards in the routine which cause the computer to correct the product or to set an alarm if the computer is requested to produce a product which does not lie within the range allowable by the floating point representation; that is, α and α' may have legitimate floating point representations, and α'' may not. For example, if $\alpha = \alpha' = +1000000076$ ($\alpha = 1.0000000 \times 10^{26}$), then the pseudo exponent of α'' is $\bar{n}'' = 76 + 76 - 50 = 102$ and $102 \geq 100$; similarly, if $\alpha = \alpha' = +1000000024$ ($\alpha = 1.0000000 \times 10^{-26}$), then the pseudo exponent of α'' is $\bar{n}'' = 24 + 24 - 50 = -02$ and $-02 < 00$.

This discussion of floating point notation, although carried out in a general form, has assumed that the base a is decimal. If $a = 16$, the discussion would probably be unchanged and if $a = 8$, the computer words would probably contain 12 digits, $\alpha = \alpha_0\alpha_1 \cdots \alpha_{10}\alpha_{11}$, and α_{10} and α_{11} would represent the pseudo exponent. If $a = 2$, the word size would probably be 40 digits (bits) and $\alpha_{32} \cdots \alpha_{39}$ would represent the pseudo exponent. The number bases, i.e., 2, 8, 10, and 16, used in existing computers are even integers, and the digit $a/2$ may be used for rounding and $a^{p/2}$ may be used for exponent modifications, where p is a positive even integer.

The process of determining the multiplication factors for each variable of a problem so that these variables and the result of operations upon them will be numbers representable by the computer is the process of *scaling* a problem, and the factors are called the *scale factors*. The ability to scale problems for a computer depends upon the ability to work with inequalities and upon a knowledge of error generation and propagation. A knowledge of the physical behavior or mathematical properties of the problem is often useful in scaling a problem. A later chapter is devoted to some of the basic concepts of numerical analysis and contains a bibliography of books and articles on numerical analysis which may be helpful. A basic knowledge of inequalities, such as contained in any elementary college algebra text, is assumed for the reader. Through examples of programming problems and subroutines, many of the methods of scaling a problem will be exhibited in this book. In floating point computations the computer automatically processes the scale factors. In fixed point computations (*cf.* next paragraph) the programmer provides for the scale factor processing. In either case the programmer must be aware of the nature of intermediate results and the possibilities for loss of significance in the results due to accumulation of rounding errors and cancellation of significant digits.

For a computer containing words of length m digits with base a, we will represent a number in such a computer as

$$\alpha = \pm .\alpha_0\alpha_1 \cdots \alpha_{m-1} = \pm \sum_{r=0}^{m-1} \alpha_r a^{-r-1} \tag{2-4-2}$$

(Note that in this representation α_0 is the coefficient of a^{-1}, α_1 is the coefficient of a^{-2}, etc., whereas in Eqs. (2-3-6) and (2-4-1), α_0 is the coefficient of a^0, α_1 is the coefficient of a^{-1}, etc. Both representations are used in computer manuals.) Such a representation is referred to as a *fixed point representation*, and the number for such a representation lies in the range $-1 < \alpha < 1$. The particular choice of Eq. (2-4-2) for the representation of allowable numbers of the computer will be discussed after we have described the arithmetic unit of the computer.

Suppose we add two numbers α and α', of the form of Eq. (2-4-2) together—remembering that the sum $\alpha'' = \alpha + \alpha'$ must be storable in the memory in the form of Eq. (2-4-2)—then we immediately realize that α and α' must be such that the sum lies in the range $-1 < \alpha'' < 1$. The register in the arithmetic unit of the computer, in which the results of addition, subtraction, multiplication, and the remainder of division are formed, is called the *accumulator* (abbreviated A), and contains the sign and the digit positions $A_{-1}A_0 \cdots A_{m-1}A_m \cdots A_{2m-1}$. This accumulator is often divided into two parts: the *upper* or *left accumulator* (abbreviated A_U) containing the sign and the digit positions $A_{-1}A_0 \cdots A_{m-1}$ and the *lower* or *right accumulator* (abbreviated A_L) containing the digit positions $A_m \cdots A_{2m-1}$. By having the result α'' of the addition $\alpha + \alpha'$ appear in the upper accumulator, A_{-1} may be checked automatically by the computer to determine if it contains a one or a zero before α'' is stored in the memory. If in the addition of two numbers of like sign A_{-1} contains a one, then c_0, the carry digit from $\alpha_0 + \alpha_0' + c_1$, is 1 and $|\alpha''| \geq 1$. If A_{-1} contains a 0, then $|\alpha''| < 1$. Whenever A_{-1} of the upper accumulator contains a one at the end of an arithmetic operation, an *overflow* is said to have occurred. By having the computer stop before the execution of the next order after an overflow, except when the next order contains a "transfer on overflow" command, the computer can assist in detecting fallacious calculations. A transfer on overflow command for the three-address computer of Chap. 1 is as follows:

OVW $M_1M_2M_3 \Rightarrow$ Ignore M_1 and M_2. If an overflow occurred as the result of the preceding order, take the next order from memory location M_3. If an overflow did not occur, take the next order sequentially.

With such a command the result of an addition can be checked, and if overflow has occurred, then the computer may take its next order from a subroutine[1] which, instead of performing $\alpha + \alpha'$, performs $a^{-1}(\alpha + \alpha') = a^{-1}\alpha + a^{-1}\alpha' = a^{-1}\alpha''$ and obtains an answer within the range used by the computer. For example, suppose on a computer for which a and m are 10 we wish as part of a problem to perform the calculation $\alpha'' = .00035875\ (\alpha + \alpha')$, where $-1 < \alpha,\ \alpha' < 1$; then, for many combinations of α and α', this subroutine could be carried out as in the flow diagram of Fig. 2-4-2 where

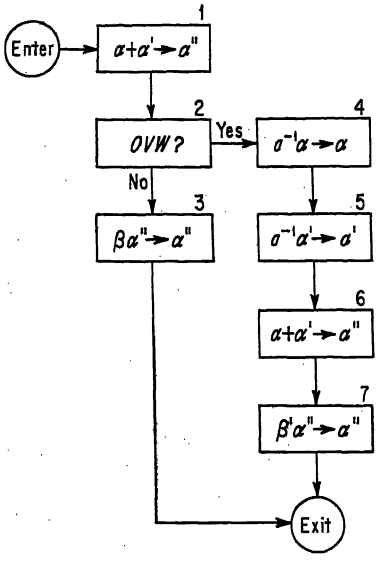


FIG. 2-4-2. A flow diagram exhibiting the use of the OVW order.

$$\beta = +.0003587500,$$
$$\beta' = +.0035875000$$

and OVW? implies the command OVW $M_1M_2M_3$. Since

$$\alpha'' = .00035875\ (\alpha + \alpha')$$

the answer will be a number within the computer's range as long as α and α' are numbers within range. However, it may be that $|\alpha + \alpha'| \geq 1$, in which case an overflow will occur before α'' is obtained. Therefore, in box 2 of Fig. 2-4-2 the addition is checked to see if the condition $|\alpha + \alpha'| < 1$ is satisfied. If an overflow has occurred, then $\beta(\alpha + \alpha')$ is obtained by forming

$$10\beta(10^{-1}\alpha + 10^{-1}\alpha') = \beta'(10^{-1}\alpha + 10^{-1}\alpha')$$

It may be noted that in the above process, one digit of precision is lost during the scaling for the overflow case.

As previously described, subtraction is carried out in the arithmetic unit by changing the sign of the subtrahend and adding. For multiplication and division, another register, which is usually called the *quotient register* (abbreviated Q), is added to the arithmetic unit. This register contains a sign and the digits $q_{-1}.q_0q_1 \cdots q_{m-1}$.

[1] A subroutine is a part of a computer program. It is usually designed to perform one or more computation or input-output tasks, *cf.* Chapter 5.

For the three-address multiply command MLT $M_1M_2M_3$, let the contents of memory location M_1 be α and the contents of M_2 be α'. For such an order, the accumulator is initially set to zero

$$(A_{-1}) = (A_0) = \cdots = (A_{2m-1}) = 0$$

and $|\alpha'|$ is sent to Q, the digits α'_0 through α'_{m-1} going to q_0 through q_{m-1} respectively. The digit position q_{m-1} is tested for zero. If $q_{m-1} \neq 0$, then q_{m-1} is decreased by one, and $|\alpha|$ is added to the upper accumulator in digit positions A_0 through A_{m-1}. This procedure is repeated with carry digits carrying over into A_{-1} until $q_{m-1} = 0$. When $q_{m-1} = 0$, then $|\alpha|$ has been added to the upper accumulator α'_{m-1} times. Next, the contents of the accumulator, upper and lower, and the contents of Q are shifted right one digit, and a zero is inserted at the left end of A and Q. The contents of Q are now the digits $0\alpha'_0\alpha'_1 \cdots \alpha'_{m-2}$ in the positions $q_0q_1q_2 \cdots q_{m-1}$ respectively. The previous process is repeated; that is, $|\alpha|$ is added to the accumulator, in digit positions A_0 through A_{m-1}, α'_{m-2} times. This process of adding $|\alpha|$ to the accumulator α'_r times and shifting right one digit is continued until $|\alpha|$ is added to the upper accumulator α'_0 times. Then the contents of the accumulator, upper and lower, are shifted one digit to the right. The digit positions, $A_0 \cdots A_{m-1}A_m \cdots A_{2m-1}$, of the accumulator now contain the product $\alpha\alpha'$ and A_{-1} contains a zero. The most significant digits of $\alpha\alpha'$ are in the upper accumulator, and the least significant digits are in the lower accumulator. We note that an overflow will not occur since if $|\alpha| < 1$ and $|\alpha'| < 1$, then $|\alpha\alpha'| < 1$. The sign of the product is inserted in the sign digit of A_U. Since the $2m$ digits of the product are available, we add the multiplication order:

MLT $M_1M_2M_3 \Rightarrow$ Store the most significant half of the digits of the product m_1m_2 in memory location M_3 and the least significant half of the digits of m_1m_2 in memory location $M_3 + 1$. The accumulator sign is common to A_U and A_L.

Further, since the product of m_1, the word in memory location M_1, and m_2, the word in memory location M_2, appears in the digit positions $A_0 \cdots A_{m-1}A_m \cdots A_{2m-1}$, then, before storing the product in the memory, $(a/2)a^{-m-1}$ may be added to the magnitude in the accumulator, producing thereby a rounded possible product in the digit positions $A_0 \cdots A_{m-1}$. Examination of the largest product shows that this process cannot pro-

duce an overflow. The following multiplication command of Chap. 1 called "multiply round" is repeated:

MLR $M_1M_2M_3 \Rightarrow m_1 \times m_2 = m_3$ where m_3 is the rounded product.

The function of the accumulator and the quotient register will now be exhibited in connection with the division command DIV $M_1M_2M_3$ for the three-address computer. Again let α be the word stored in memory location M_1, and α' be the word stored in memory location M_2. It is desired to compute $\alpha'' = \alpha/\alpha'$ where α and α' are of the form of Eq. (2-4-2) and the result α'' is also to be stored in memory in this form. That is, we are given $|\alpha| < 1$ and $|\alpha'| < 1$, and it is desired to form α'' whenever $|\alpha''| < 1$, i.e., whenever $|\alpha| < |\alpha'|$, and to stop the computer (or set the overflow toggle at the completion of the division command) if $|\alpha''| \geq 1$.

The division procedure as carried out in the arithmetic unit is to clear the accumulator and enter $|\alpha|$ in the upper accumulator in digit positions A_0 through A_{m-1} and then subtract $|\alpha'|$ from the upper accumulator. If $|\alpha| - |\alpha'| \geq 0$, then stop the computer (or set the overflow toggle) because $|\alpha| \geq |\alpha'|$ and $|\alpha''|$ will be greater than or equal to one. It may be noted that this test will stop the computer whenever the divisor is zero, even in the indeterminant case when both dividend and divisor are zero. If $|\alpha| - |\alpha'| < 0$, add $|\alpha'|$ to A_U and shift the contents of the accumulator left one digit position. The accumulator now contains the digits α_0 through α_{m-1} in the digit positions A_{-1} through A_{m-2} and zeros in the digit positions A_{m-1} through A_{2m-1}. Subtract $|\alpha'|$ from A_U and test to see if $(A) \geq 0$. If $(A) \geq 0$, add $a^{-m} = +.00 \cdots 01$ to Q, subtract $|\alpha'|$ from A_U, and again test to see if $(A) \geq 0$. This procedure is continued until $(A) < 0$. When $(A) < 0$, add $|\alpha'|$ to A_U and shift the contents of A and of Q one digit position to the left. This process is repeated through $m - 1$ shifts of Q. After $(A) < 0$ and $|\alpha'|$ is added to A_U, but before the mth shift left of one digit position of Q, the division process may be stopped. The digits of the quotient α'', i.e., α_0'' through α_{m-1}'', now appear in Q in the digit positions q_0 through q_{m-1} respectively. The sign affixed to α'' is plus if the signs of α and α' are the same, and it is minus if the signs of α and α' are different. At this time A_U contains the absolute value of the remainder times a^m. The sign to be attached to the remainder is always the same as that of the dividend α. Since the quotient and a^m times the remainder are available, the division order can be changed to:

DIV $M_1M_2M_3 \Rightarrow m_1 \div m_2 = m_3$ where the quotient is stored in memory location M_3 and a^m times the remainder is stored in memory location $M_3 + 1$.

Since the digit position q_{-1} of Q is still available, the mth shift left of Q and the $(m + 1)$th shift left of A of the previous procedure could be performed, and the process stopped before the $(m + 1)$th shift of Q takes place. At this time Q contains α_0'' through α_{m-1}'' in the digit positions q_{-1} through q_{m-2} respectively, and the next digit of the quotient in digit position q_{m-1}. By adding $a/2$ in the digit position q_{m-1} and shifting the contents of Q right one digit position, Q contains the rounded quotient of $|\alpha|/|\alpha'|$ in digit positions q_0 through q_{m-1} respectively. The sign of the quotient is inserted as before. The following division command called "divide round" is repeated for the three-address computer of Chap. 1:

DVR $M_1M_2M_3 \Rightarrow m_1 \div m_2 = m_3$ where m_3 is the rounded quotient.

Continuing with the *fixed point* representation of numbers, it can be shown that if the length of a base-a computer word is m digits and sign, then the computer can distinguish between $2a^m - 1$ different numbers. The "2" in the preceding number allows distinguishing between positive and negative numbers and the "-1" exists because we do not wish the computer to recognize $+0$ as a different number from -0. The precision, then, of the computer in differentiating between two numbers is one part in $(2a^m - 1)$ parts. Thus, independent of the magnitude of the words handled by a computer (the placement of the radix point), the precision of the computer remains the same.

Since the addition of two numbers in a computer is performed by adding, in A_U, the m digits of one word to the m digits of the other word, the sum of the two words can be carried out correctly only if their radix points align. The sum may be stored in memory if an overflow does not occur. If two numbers are stored in memory with the radix points in different positions then one number must be shifted in the accumulator to align the radix points before addition or subtraction can be performed correctly by the computer.

Most computers form the product of two numbers α and α' by a procedure equivalent to adding $|\alpha|$ to A_U α'_{m-1} times, shifting the contents of the accumulator one digit to the right, adding $|\alpha|$ to A_U α'_{m-2} times, and continuing this process until $|\alpha|$ has been added to A_U, α'_0 (the leftmost digit of $|\alpha'|$) times, and the contents of the accumulator is shifted one digit position to the right. The sign of the product $\alpha\alpha'$ is assigned to the accumulator at the end of the multiplication command. Thus, the m most significant digits of $\alpha\alpha'$ appear in A_U and the m least significant digits of $\alpha\alpha'$ appear in A_L. If s and t are two integers such that $-1 \leq s,\ t \leq m - 1$, and if the radix point of α is located between the digits α_s and α_{s+1} and the radix point of α' is located between α'_t and

α'_{s+1}, then the radix point of the product $\alpha\alpha'$ is located between digit positions A_{s+t+1} and A_{s+t+2} of the accumulator. If $s + t = 2m - 2$, then the radix point is located at the right end of A.

Similarly, in the division of α by α', the location of the radix point of the quotient of two numbers depends upon the location of the radix points of the dividend and divisor. With s and t designating, as before, the locations of the radix point in α and α' respectively, the radix point for the quotient is located between q_{s-t-1} and q_{s-t}.

Thus, it is apparent that we can locate the radix point anywhere we wish in a word, but we must be careful in programming problems for the computer to keep a record of the location of the radix point before and after each arithmetic operation, and the radix points of two words must be lined up before adding one to the other, before subtracting one from the other, or comparing one with the other. However, as we have already demonstrated, by having the words represented as in Eq. (2-4-2), the radix point may be considered as automatically located at the left of α_s by the computer, and the range of the variables may be handled by introducing scale factors into the equations. Thus, in the remainder of this text we will assume numbers to have this fixed point representation.

Let us now return to the example of obtaining the product $\alpha'' = \alpha\alpha'$ where $\alpha > 0$ and $\alpha' > 0$ and α, α' and α'' are in floating point notation. However, let us write the pseudo exponent as the first two digits of the number. In this example we use decimal representation and a 10-digit-and-sign word length. Let

$$\bar{n} = \alpha_s\alpha_1.00000000 \qquad \bar{n}' = \alpha'_0\alpha'_1.00000000 \qquad \bar{n}'' = \alpha''_0\alpha''_1.00000000$$

and

$$\beta = +00.\alpha_2\alpha_3 \cdots \alpha_9 \qquad \beta' = +00.\alpha'_2\alpha'_3 \cdots \alpha'_9$$

$$\beta'' = +00.\alpha''_2\alpha''_3 \cdots \alpha''_9$$

where $$a^{-1} \leq \beta, \beta', \beta'' < 1, \quad a = 10$$

then

$$\alpha = 10^{\bar{n}-50}\beta = a^{\bar{n}-a^2/2}\sum_{r=2}^{9}\alpha_r a^{1-r}$$

$$\alpha' = 10^{\bar{n}'-50}\beta' = a^{\bar{n}'-a^2/2}\sum_{r=2}^{9}\alpha'_r a^{1-r} \qquad (2\text{-}4\text{-}3)$$

and

$$\alpha'' = 10^{\bar{n}''-50}\beta = a^{\bar{n}''-a^2/2}\sum_{r=2}^{9}\alpha''_r a^{1-r}$$

Again, the number zero is assumed to have the floating point representation in which $\alpha_0 = \alpha_1 = \cdots = \alpha_9 = 0$. By using the floating point

representation of Eq. (2-4-3), where the word α contains the ten digits α_0 through α_9, the TRA command may be used directly to determine which of two floating point numbers is the greater. However, each number represented by Eq. (2-4-1), in which the pseudo exponent occurs at the right, must be divided into two numbers, one containing the sign of the word and the pseudo exponent and the other containing the sign of the word and the precision digits, before a comparison of two such numbers can be made. If the sign and pseudo exponent of one number are algebraically greater than the sign and pseudo exponent of the other, the comparison is completed; if they are the same, then the signs and the precision digits of the two numbers must also be compared. The floating point representation of Eq. (2-4-1) was included because of its wide use in the past.

Returning to the product $\alpha'' = \alpha\alpha'$ where the representation for each number is defined by Eq. (2-4-3), we now discuss the computer computation for products in this form. Let

$$\gamma = a^{-2}\beta = +.00\alpha_2\alpha_3 \cdots \alpha_9$$

and

$$\gamma' = a^{-2}\beta' = +.00\alpha'_2\alpha'_3 \cdots \alpha'_9$$

then γ and γ' are computer words with fixed point representations as defined by Eq. (2-4-2). Define

$$\gamma'' = (a^2\gamma)(a\gamma') = .0\gamma''_1\gamma''_2 \cdots \gamma''_9$$

where $0, \gamma''_1, \gamma''_2, \ldots, \gamma''_9$ are the leading digits of the product $a^3\gamma\gamma'$, then

$$\beta'' = \begin{cases} a^2\gamma'' = 00.\gamma''_2\gamma''_3 \cdots \gamma''_9 & \text{if } \gamma''_1 = 0 \\ a\gamma'' = 00.\gamma''_1\gamma''_2 \cdots \gamma''_8 & \text{if } \gamma''_1 \neq 0 \end{cases}$$

and

$$\bar{n}'' = \begin{cases} \bar{n} + \bar{n}' - 51 & \text{if } \gamma''_1 = 0 \\ \bar{n} + \bar{n}' - 50 & \text{if } \gamma''_1 \neq 0 \end{cases}$$

Thus, $\alpha'' = 10^{\bar{n}''-50}\beta''$ and possesses a floating point representation in the form of Eq. (2-4-3) when $0 \leq \bar{n}'' < 100$. Addition, subtraction, and division of numbers in floating point representation can be discussed in a similar fashion.

We add the following floating point arithmetic commands to the three-address computer:

FAD $M_1M_2M_3$ $\Rightarrow m_3 = m_1 + m_2$ where m_1, m_2, and m_3 are in floating point notation.

FSB $M_1M_2M_3$ $\Rightarrow m_3 = m_1 - m_2$ where m_1, m_2, and m_3 are in floating point notation.

FMR $M_1M_2M_3 \Rightarrow m_3 = m_1m_2$ where m_3 is the rounded product and m_1, m_2, and m_3 are in floating point notation.

FDR $M_1M_2M_3 \Rightarrow m_3 = m_1/m_2$ where m_3 is the rounded quotient and m_1, m_2, and m_3 are in floating point notation.

In these orders we assume the floating point notation is that of Eq. (2-4-3).

2-5. Number Representation Conversion

Often data to be processed by a computer are given in a number base representation other than that used by the computer. This is especially true for binary computers where much of the input data to be processed is in the decimal representation. Usually, binary codes are automatically assigned to the decimal digits so that the digital information may be read into the computer; and, then, a computer conversion routine is used to convert the *binary-coded decimal data* to binary representation. Similarly, the binary representation of computational results obtained by a binary computer are often converted (by the binary computer) to a binary-coded decimal representation so that they may be examined by persons not familiar with the binary representation of numbers. Procedures for converting a word in one number representation to another number representation are outlined in the remainder of this chapter. It will be assumed that all number representations have positive integer bases greater than one.

The process of converting input data to the number representation of the computer is a relatively simple procedure. Let us assume that a number is given in base d as

$$\delta = \sum_{n=0}^{N} \delta_n d^{-(n+1)} \tag{2-5-1}$$

and that the computer uses base b arithmetic. Further, we assume that there is a unit which automatically codes each digit δ_n of the base d word as a number in base-b representation. That is,

$$\delta_n = \sum_{p=0}^{P} \beta_{np} b^p \tag{2-5-2}$$

where the β_{np} are base-b digits. Table 2-2-1 may be used to code base-8, -10, or -16 digits to base-2 digits. Thus, we have a means of storing the base-d digits in the base-b computer. However, this need not represent

the conversion of the base-d word to a base-b word. Consider the number $(0.75)_{10}$. The digit $7_{10} = (111)_2$ and the digit $5_{10} = (101)_2$; however

$$(0.75)_{10} \neq (0.111101)_2$$

since $(0.111101)_2 = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{64} = (0.953125)_{10}$
On the other hand, as shown in Sec. 3-9,

$$(0.75)_8 = 7 \times 8^{-1} + 5 \times 8^{-2}$$
$$= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{64} = (0.953125)_{10}$$

To understand how the conversion may be properly achieved, write Eq. (2-5-1) as

$$\delta = \sum_{n=0}^{N} \delta_n d^{-(n+1)} = \{ \cdots [(\delta_N d^{-1} + \delta_{N-1}) d^{-1} + \delta_{N-2}] d^{-1} + \cdots + \delta_0 \} d^{-1} \quad (2\text{-}5\text{-}3)$$

By also having $d^{-1} < 1$ in base b, i.e.,

$$d^{-1} = \sum_{r=0}^{R} \beta_r b^{-(r+1)} \quad (2\text{-}5\text{-}4)$$

we may substitute Eqs. (2-5-2) and (2-5-4) in Eq. (2-5-3) giving the desired conversion formula

$$\delta = \Big\{ \cdots \Big[\Big(\sum_{p=0}^{P} \beta_{N,p} b^p \times \sum_{r=0}^{R} \beta_r b^{-(r+1)} + \sum_{p=0}^{P} \beta_{N-1,p} b^p \Big) \sum_{r=0}^{R} \beta_r b^{-(r+1)} + \sum_{p=0}^{P} \beta_{N-2,p} b^p \Big] \sum_{r=0}^{R} \beta_r b^{-(r+1)} + \cdots + \sum_{p=0}^{P} \beta_{0,p} b^p \Big\} \sum_{r=0}^{R} \beta_r b^{-(r+1)} \quad (2\text{-}5\text{-}5)$$

Returning to our example,

$$(\tfrac{1}{10})_{10} \doteq (0.000110011)_2$$

thus, by Eq. (2-5-5)

$$(0.75)_{10} \doteq [(101.0)(0.000110011) + 111.0](0.000110011)$$
$$\doteq (0.10111111)_2 = (0.74609375)_{10}$$

with a truncation error of 2^{-8}. Actually, $(0.75)_{10} = (\frac{3}{4})_{10} = (0.11)_2$.
Similarly, $(\frac{1}{8})_{10} = (0.001)_2$ and by Eq. (2-5-5)

$$(0.75)_8 = [(101.0)(0.001) + 111.0](0.001) = (0.111101)_2$$

Using a computer to convert results to a number representation with a base other than its own implies that the digits of the other representation

may be coded in the representation used by the computer. Conversion routines are often written for binary computers which convert numbers in the binary representation to a binary-coded decimal representation. That is, four binary digits are used to represent a single decimal digit in accordance with Table 2-2-1. For example, the binary number 0.11 is converted to the number 0000. 0111 0101 in binary-coded decimal. The first four digits to the right of the decimal point, 0111, are interpreted as 7 and the last four digits are interpreted as 5. Thus, the conversion problem from binary to decimal representation may be considered to be that of translating from binary representation to binary-coded decimal representation. The following discussion, then, is for the conversion from a number representation in base b to one in base d; b and d are positive integer bases greater than 1, where the conversion is done with base b arithmetic. Our discussion depends on the following lemma and two definitions.

LEMMA. If two finite number representations represent the same number, their integral parts are equal, and their fractional parts are equal.

DEFINITIONS.

1. For $x \geq 0$, $[x]$ represents the largest integer not exceeding x.
2. $\langle x \rangle = x - [x]$.

The integral part of x is $[x]$, and the fractional part of x is $\langle x \rangle$.

We assume that we are given a positive number with representation

$$\beta = \sum_{r=0}^{R} \beta_r b^{-(r+1)}$$

and we desire the representation

$$(\beta)_d \doteq \delta = \sum_{n=0}^{N} \delta_n d^{-(n+1)}$$

where the approximation occurs since we desire only a finite number of digits N in the representation δ. Thus, let us write

$$\sum_{n=0}^{N} \delta_n d^{-(n+1)} \doteq \sum_{r=0}^{R} \beta_r b^{-(r+1)} \qquad (2\text{-}5\text{-}6)$$

and, by multiplying both sides of Eq. (2-5-6) by d, we have

$$\delta_0 + \sum_{n=1}^{N} \delta_n d^{-n} = (d)_b \sum_{r=0}^{R} \beta_r b^{-(r+1)} \qquad (2\text{-}5\text{-}7)$$

Applying the lemma, Eq. (2-5-7) gives

$$\delta_0 = \left[(d)_b \sum_{r=0}^{R} \beta_r b^{-(r+1)} \right] \tag{2-5-8}$$

and

$$\sum_{n=1}^{N} \delta_n d^{-n} \doteq \left\langle (d)_b \sum_{r=0}^{R} \beta_r b^{-(r+1)} \right\rangle \tag{2-5-9}$$

Next, we repeat the process and multiply both sides of Eq. (2-5-9) by d giving

$$\delta_1 + \sum_{n=2}^{N} \delta_n d^{-n+1} \doteq (d)_b \left\langle (d)_b \sum_{r=0}^{R} \beta_r b^{-(r+1)} \right\rangle \tag{2-5-10}$$

Applying the lemma to Eq. (2-5-10) gives

$$\delta_1 = \left[(d)_b \left\langle (d)_b \sum_{r=0}^{R} \beta_r b^{-(r+1)} \right\rangle \right] \tag{2-5-11}$$

and

$$\sum_{n=2}^{N} \delta_n d^{-n+1} \doteq \left\langle (d)_b \left\langle (d)_b \sum_{r=0}^{R} \beta_r b^{-(r+1)} \right\rangle \right\rangle \tag{2-5-12}$$

The preceding process is continued until the N coefficients δ_n are obtained.

Returning to our example, consider the binary number $(0.11)_2$ which we wish to convert to binary-coded decimal representation. Thus, we have

$$\beta = (1)_2 \times (0.1)_2 + (1)_2 \times (0.01)_2$$

and

$$\delta = (\delta_0)_2 \times (1010)_2{}^{-1} + (\delta_1)_2 \times (1010)_2{}^{-2}$$

since $(10)_{10} = (1010)_2$. By Eq. (2-5-8), we have

$$\begin{aligned} (\delta_0)_2 &= [(1010)_2 \times (0.11)_2] = [(111.1)_2] \\ &= (0111)_2 \end{aligned}$$

and thus $\delta_0 = 7$. Similarly, by Eq. (2-5-11)

$$\begin{aligned} (\delta_1)_2 &= [(1010)_2 \times \langle (111.1)_2 \rangle] = [(1010)_2 \times (0.1)_2] \\ &= (0101)_2 \end{aligned}$$

and $\delta_1 = 5$. Thus, $\delta = 0.75$.

PROBLEMS

2-1. Given α and α' in the floating point notation of Eq. (2-4-3), construct a flow diagram for the operations of floating point addition and subtraction.

2-2. Describe how the multiplication of two positive numbers may be performed by successive additions without the use of the Q register. Suggestion: Send the multiplicand to A_U and the multiplier to A_L.

2-3. Describe how the division of two positive numbers may be performed without the use of the Q register.

2-4. For what range of numbers α and α' will α'', in the problem of the flow diagram of Fig. 2-4-2, be zero?

2-5. Give a proof for the lemma of Sec. 2-5.

2-6. Prove that the coefficients δ_n as acquired by Eqs. (2-5-9), (2-5-11), etc., are single digits in base-d representation.

2-7. Develop a procedure for converting from a base-d representation to a base-b representation for an integer.

3

PROGRAMMING AND CODING

3-1. Introduction

In Chap. 1, the reader was introduced to programming and coding through a discussion which demonstrated the need for a flow diagram. This flow diagram was used to show the organization of a problem and to exhibit the sequencing of the parts of the problem so that it might be coded for an automatic digital computer. In this chapter, a more detailed discussion of the procedures for drawing flow diagrams and the deriving of a code from them will be given. At first two types of flow diagrams, the problem flow diagram and the computer flow diagram, will be introduced as an aid to learning. In deriving a code from the computer flow diagram, codes for a three-address and a two-address computer will be used. The word size of these computers will be assumed to be ten decimal digits and a sign digit. The magnitude of the number representations is assumed to be in the range $0 \leq |w| < 1$.

After several codes have been written for each computer, we will introduce a detailed flow diagram, the object of which is, insofar as possible, to relieve the programmer of drawing both a problem and a computer flow diagram. In the last section of this chapter some of the concepts of coding for binary computers are introduced.

Much of the material covered in this chapter stems from the investigations of John von Neumann, H. H. Goldstine, and A. W. Burks as given in their U.S. Army Ordnance reports[1] and the notes[2] on lectures given by D. A. Flanders at Oak Ridge National Laboratory.

[1] See footnote 1 to the preface.

[2] D. A. Flanders, "Notes on an Introduction to the Programming and Coding of Problems for the ORNL Electronic Digital Automatic Computer," Argonne National Laboratory, Dec. 3, 1951.

3-2. The Problem Flow Diagram

The problem flow diagram is, as the term indicates, a road map or flow chart of the problem. The problem, however, should be properly stated for the computer before such a diagram is drawn. For example, the problem of finding the sum s given by Eq. (1-4-1)

$$s = \sum_{j=1}^{n} a_j \qquad (1\text{-}4\text{-}1)$$

of Chap. 1 is symbolically described by that equation. But, such a description must be enlarged before it is adequate for computation. To complete the description for a computer we need to know an upper bound for n, an upper bound for the magnitude of all a_j, and the desired accuracy for the sum. Suppose these requirements are as follows:

$$n \leq 700$$
$$|a_j| < 1$$

and, instead of directly specifying the accuracy for the computation of the sum s, we stipulate that each a_j may be rounded to the seventh digit to the right of the decimal point. From these requirements, s will be less than 1,000 but may be greater than 100. The scaled problem for the computer, then, is that of calculating

$$\bar{s} = \sum_{j=1}^{n} \bar{a}_j \doteq 10^{-3}s \qquad (3\text{-}2\text{-}1)$$

where

$$\bar{a}_j \doteq \begin{cases} 10^{-2}(10^{-1}a_j + r) & a_j \geq 0 \\ 10^{-2}(10^{-1}a_j - r) & a_j < 0 \end{cases} \qquad (3\text{-}2\text{-}2)^1$$

and

$$r = 5 \times 10^{-9} \qquad (3\text{-}2\text{-}3)$$

Here, r is used to produce a rounded value for each $\bar{a}_j$. The powers of 10 are used to indicate shifts of the digits. Digits shifted out of the word are lost. (If a rounded value for $\bar{a}_j$ were produced by adding 5×10^{-8} to $|a_j|$, then an overflow would occur whenever $|a_j| \geq 1 - 5 \times 10^{-8}$.) Thus, $\bar{s} \doteq 10^{-3}s$ and satisfies the condition

$$|\bar{s}| < 1$$

which is the restriction on the magnitude of a computer word. Restating: our computer problem is that of calculating $\bar{s}$ by Eqs. (3-2-1), (3-2-2), and (3-2-3) from a given set of a_j.

[1] This scale and round operation can be performed with one command if the computer has a shift and round operation or if the multiply and round command is used.

For the problem flow diagram we will use the symbols introduced in Sec. 1-4, except for the arrow within a box. Each box will contain either an equation, an inequality, a statement in words, or a combination of these. The dynamic nature of computation processes, particularly the interactions in a computation, should be kept in mind, i.e., parts may undergo modification during the computation, and thus the definition of terms may depend on the preceding computation steps. The flow diagram illustrates these interactions in the computation. The use of the circle in the flow diagram will be extended to that of a *connector*. In addition to indicating where a problem begins or ends, connectors are used to show linkage between boxes. That is, instead of joining two boxes by an extended arrow we may, as shown in Fig. 3-2-1, choose to draw an arrow from one box to a circle and draw another arrow from another circle to the second box. Each of the two circles contains the same identification symbol, usually a number or a Greek letter. . Such connectors, which indicate that the flow diagram leaves one box and always enters a second, are referred to as *fixed connectors*. In this figure the symbol within the circle was chosen to be the box number of the second box. Often there will be several possible exits from one box, the choice being determined by the computer in the course of computation. One may, as shown in Fig. 3-2-2, attach a generic symbol, e.g., α_i, as the common exit symbol and use particular values of i, e.g., α_1, α_2, . . . , for the various alternate entries. Such an exit connector is referred to as a *variable connector*.

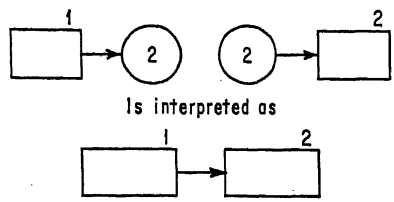


FIG. 3-2-1. Fixed connector.

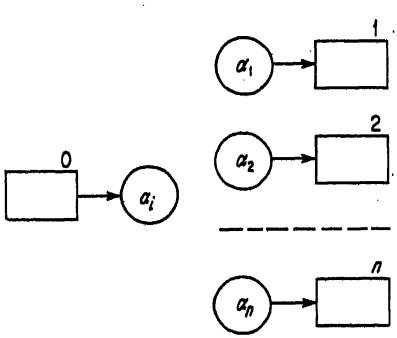
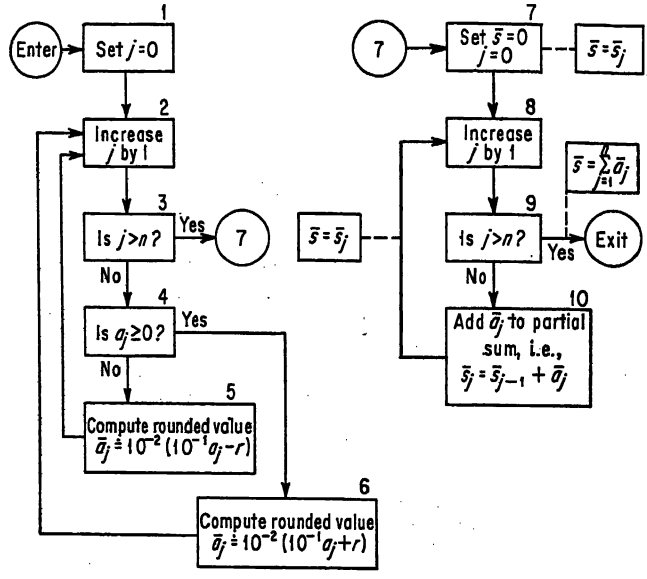


FIG. 3-2-2. Variable connector.

The problem flow diagram for the problem of Eqs. (3-2-1), (3-2-2), and (3-2-3) is given in Fig. 3-2-3. The flow diagram has been divided into two parts. The first part, boxes 1 through 6, is for the scaling and rounding of the a_j's using Eqs. (3-2-2) and (3-2-3); and the second part, boxes 7 through 10, is for the calculation of $\bar{s}$ from the $\bar{a}_j$'s using Eq. (3-2-1). The problem is split into two parts since in larger problems it is often

desirable to do scaling and arranging of input data during the loading process. Thus, boxes 1 through 6 might be considered as a part of a load routine. For a simple sum such as this, the sum would normally be taken care of at the time of scaling. In each part, the subscript j is increased by one and then compared with n before the calculations are performed.



Notes: (1) $r = 5 \cdot 10^{-9}$

(2) $\bar{s}_j = \sum_{i=1}^{j} \bar{a}_i = \bar{a}_j + \bar{s}_{j-1}$

FIG. 3-2-3. Problem flow diagram for $\bar{s} = \sum_{j=1}^{n} \bar{a}_j$.

This sequence of operations is chosen so that the flow chart will apply when $n = 0$ as well as when $n \geq 1$. (The flow diagram of Fig. 1-4-1 assumes $n \geq 1$.) At first glance, $n = 0$ appears not only to be a special but also a trivial case. However, this problem might be a part of a larger problem in which many sums of this type are being performed. Some of the sums may contain no a_j's and, rather than separate these from the rest, one designs the flow diagram so that they are computed in the same fashion as those sums containing one or more a_j's. The monthly billing of accounts is a problem of this type since some accounts may have

no activity during the month. If there is the requirement to keep track of the activity of each account, then all accounts must be examined and both n and $\bar{s}$ recorded.

In box 3 of the first part of the flow diagram of Fig. 3-2-3, j becomes greater than n when $j = n + 1$. At this time all a_j have been converted to the scaled and rounded $\bar{a}_j$'s, and the computer is ready to evaluate the sum $\bar{s}$. Thus, the "yes" branch of box 3 leads to the second part of the flow diagram. Similarly, when $j > n$ in the second part, the sum $\bar{s}$ has been totaled, and the problem completed. Boxes 7 through 10 illustrate one of the dynamic aspects of computations. The partial sum $\bar{s}$ is initially set to zero in box 7. The partial sums $\bar{s}_j$ are stored in the same cell, $M(\bar{s})$, i.e., this cell successively contains $\bar{s} = 0$, $\bar{s}_1 = \bar{a}_1$, . . . , $\bar{s} = \bar{s}_n = \sum_{j=1}^{n} \bar{a}_j$. In each execution of the operation in box 10 a new term is added to the partial sum. The assertion box (the box connected to the flow chart by the dotted line) has no operational significance and is used to assert, at the indicated place in the flow diagram, the values of relevent parameters that are defined dynamically in the flow diagram.

3-3. The Computer Flow Diagram

A *computer flow diagram* is a reinterpretation of the problem flow diagram for a particular computer. We pointed out in Chap. 1 that problems for automatic digital computers frequently involve iterative processes. For nonrepetitive problems, the effort expended in obtaining a code for the problem would be equivalent to that for solving the problem on a desk calculator, and there would be no saving in human effort and time by using a high-speed calculator. The general properties of a computer flow diagram for an iterative problem are shown in Fig. 3-3-1.[1] The initiation box may include the operations of loading the routine and data into the computer and the setting of variables to their initial values. In the calculation box, the calculations required for a single iteration are performed, and in the interrogation box a test is performed to determine if all iterations have been completed. In the substitution box, the calculation box is prepared to repeat its calculations with new data and the progress of the problem is recorded. After all iterations have been completed, the routine goes to the termination box which

[1] This is a slightly altered version of the parts of a flow diagram as given in an article by Saul Gorn, Standardized Programming Methods and Universal Coding, *J. Assoc. Computing Machinery*, vol. 4, no. 3, pp. 254–273, July, 1957.

includes those parts of the program responsible for the arrangement of results for future use and for the listing and punching of desired results. We do not wish to imply that every computer flow diagram will appear as shown in Fig. 3-3-1. The orientation of the boxes need not be the same as that of this flow diagram.[1] For example, the second part of the problem flow diagram of Fig. 3-2-3 implies the orientation shown in Fig. 3-3-2. Also, there may be more than one box for each of those shown in Fig. 3-3-1. In fact, the figure itself may be repeated in

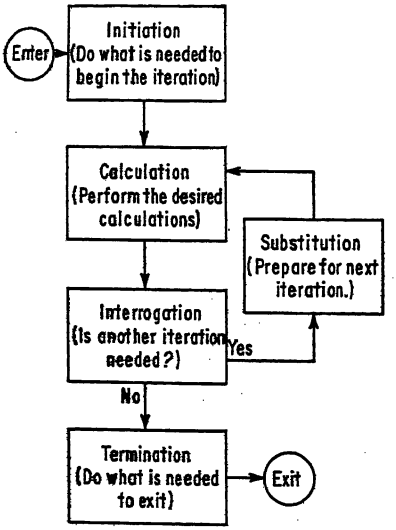


FIG. 3-3-1. Parts of a computer flow diagram.

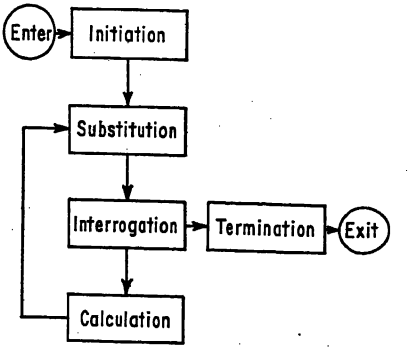


FIG. 3-3-2. A variation of the computer flow diagram.

each of the boxes. Should the problem be that of making a table of values $y_i = e^{x_i}$ where a set of x_i are given, then the basic iteration is that of replacing x_i by x_{i+1} and calculating the next y_i of the sequence. However, the calculation of e^x may be done by the Taylor series

$$e^x = \sum_{j=0}^{\infty} \frac{x^j}{j!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

and this series may also be calculated by an iterative scheme. (We remind the reader than $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$ where n is a positive integer and by definition $0! = 1! = 1$.) That is, let the terms

[1] As mentioned in Sec. 3-2, there is a definite advantage in putting the interrogation box before the calculation box since such an arrangement permits bypassing the computations if the test is satisfied *ab initio*. This is particularly important in the design of automatic compilers.

to be added be designated as

$$z_j = \frac{x^j}{j!}$$

then $$z_{j+1} = z_j \left(\frac{x}{j+1}\right)$$

where $$z_0 = 1$$

and $$e^x = \sum_{j=0}^{\infty} z_j$$

Thus, within the calculation box for the basic iteration scheme on i we have an iteration scheme on j which is shown in Fig. 3-3-3. Box 5 is used to test if the series has converged to e^x within a desired accuracy as determined by δ (for a given value of δ the truncation error is a function of x). Figure 3-3-3 is not a true problem flow diagram since we have not considered the problem thoroughly enough to determine the proper scaling of the variables.

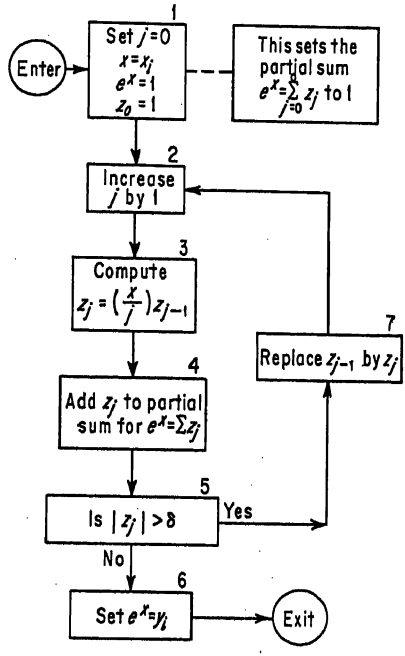


FIG. 3-3-3. Problem flow diagram for e^x.

Similarly, the initiation box may contain an iterative scheme. For example, it may be necessary to locate the data associated with the problem in several nonadjacent blocks in the memory of the computer. Thus, this data might be loaded by designating the beginning and end of the first block and filling the memory between these locations, changing the designations for the beginning and end of the block to those for the second block and filling the memory between these designated locations, etc. Iterative procedures of this type might also occur in the substitution and termination boxes. The interrogation box may also contain iterative procedures, since there may be more than one test involved in the termination of the basic iterative process.

Consider now the construction of the computer flow diagram for the fixed point calculation of the summation problem given in Eqs. (3-2-1),

(3-2-2), and (3-2-3) and outlined in the problem flow diagram of Fig. 3-2-3. For this purpose, we choose the three-address computer, the command list of which is given in Appendix I. We will use only those orders which have already been discussed and one of the two following. The first is that of adding the absolute value of a number to the absolute value of another number. The mnemonic instruction code for this order is ADA and it is defined as follows:

ADA $M_1M_2M_2 \Rightarrow m_3 = |m_1| + |m_2|$, i.e., add the absolute value of the word addressed M_1 to the absolute value of the word addressed M_2 and store the sum in memory location M_3.

Similarly, a command for subtracting the absolute value of a number from the absolute value of another number may be defined

SBA $M_1M_2M_3 \Rightarrow m_3 = |m_1| - |m_2|$.

The problem of Eqs. (3-2-1), (3-2-2), and (3-2-3) may be restated by replacing Eq. (3-2-2) by

$$\bar{a}_j \doteq \begin{cases} 10^{-2}(10^{-1}|a_j| + r) & a_j \geq 0 \\ -10^{-2}(10^{-1}|a_j| + r) & a_j < 0 \end{cases} \qquad (3\text{-}3\text{-}1)$$

and replacing the first part (boxes 1 through 6) of the problem flow diagram of Fig. 3-2-3 by Fig. 3-3-4. The advantage of this flow diagram over the one it replaces is that the basic rounding computation, box 3, appears only once.

In drawing computer flow diagrams, we place a single computer operation in each box. Also, it is often advantageous to begin the construction of the drawing at a place other than the beginning of the flow diagram. We will do this for the preceding problem and explain why afterwards. We begin by constructing the computer flow diagram for the flow diagram of Fig. 3-3-4. Box 3 of Fig. 3-3-4 becomes boxes 3.0 through 3.2 of Fig. 3-3-5. (In the drawing of computer flow diagrams we return to the use of arrows within boxes since, in general, the operation within the box indicates that the contents of a register or memory cell is being replaced by the result of a computer operation.) The three-address computer does not form the absolute value of a number directly. Thus $10^{-1}a_j$ is stored in a temporary location, the contents of which are designated by a^* (indicated in box 3.0); and, then, the sum of the abso-

lute values of a^* and $r = 5 \times 10^{-9}$ replaces a^* as shown in box 3.1. The operation of box 3.0 is enclosed in braces to indicate that it involves a variable, i.e., a_j, the address of which changes for each iteration. In box 3.2, a^* is shifted right the two remaining digit positions. This completes the rounding process. In numbering the boxes of the computer flow diagram two numbers are used for each box. The number in front of the decimal point is the number of the problem flow diagram box of which the computer flow diagram box is a part. The number after the decimal point indicates the sequence of the boxes of the computer flow diagram arising from a particular box of the problem flow diagram.

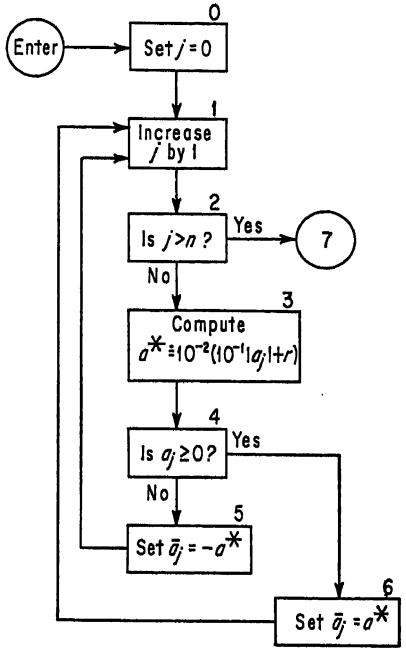


FIG. 3-3-4. Alternative problem flow diagram for first part of the flow diagram of Fig. 3-2-3.

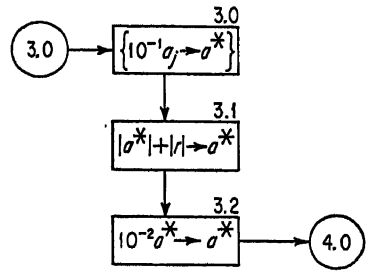


FIG. 3-3-5. Computer flow diagram for box 3 of Fig. 3-3-4.

Boxes 4, 5, and 6 of Fig. 3-3-4 are easily transformed into a computer flow diagram and are shown in Fig. 3-3-6. In constructing the computer flow diagram for box 1, we discover why the computer flow diagram for boxes 3, 4, 5, and 6 of Fig. 3-3-4 were constructed first. Box 1 of the problem flow diagram not only implies that j itself is incremented by one for each pass

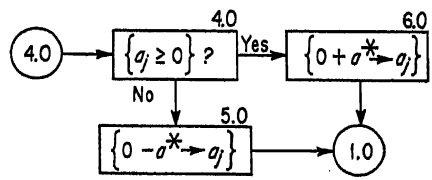


FIG. 3-3-6. Computer flow diagram for boxes 4, 5, and 6 of Fig. 3-3-4.

through that box, but also that the addresses of variables with subscript j are modified. Thus, the orders for boxes 3.0, 4.0, 5.0, and 6.0, which were enclosed in braces, will be modified by the operations which will replace box 1 in the problem flow diagram. For this purpose we list the commands of boxes 3.0, 4.0, 5.0, and 6.0 as follows:

Box 3.0:	SHR	A_j	001	A^*
Box 4.0:	TRA	A_j	0	$C_{6.0}$
Box 5.0:	SUB	0	A^*	A_j
Box 6.0:	ADD	0	A^*	A_j

where 0 is the address of the number zero, A_j is the address of a_j, A^* is the address of a^*, and $C_{6.0}$ is the address of the order for box 6.0. Again, as in Chap. 1, the a_j's are assumed to be stored sequentially in the memory with a_1 at location 300. To modify the command for box 3.0 we construct three words. The first is d_1 located at D_1 (address to be determined during coding) where

$$d_1 = \text{SHR} \quad A_1 - 1 \quad 001 \quad A^*$$

The second is d_2 located at D_2 where

$$d_2 = +0\ 001\ 000\ 000$$

and the third is j_1 located at J_1 and initially is

$$j_1 = +0\ 000\ 000\ 000$$

The order for box 3.0 is manufactured by the computer program. This construction is shown in the flow diagram of Fig. 3-3-7. In box 1.0, j_1 is

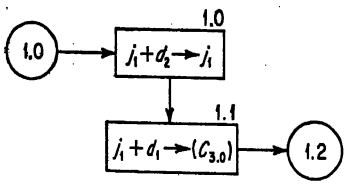


FIG. 3-3-7. Computer flow diagram for modifying the command of box 3.0.

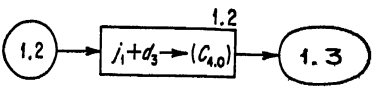


FIG. 3-3-8. Computer flow diagram for modifying the command of box 4.0.

incremented; and in box 1.1, the order for box 3.0 is obtained by adding j_1 to d_1.

By using a word d_3 located at D_3 where

$$d_3 = \text{TRA} \quad A_1 - 1 \quad 0 \quad C_{6.0}$$

we may construct a flow diagram for the modification of the command for box 4.0. This is given in Fig. 3-3-8. Similarly, by using the words

$$\begin{array}{llllll} d_4 = & \text{SUB} & 0 & A^* & A_1 - 1 \\ d_5 = & +0 & 000 & 000 & 001 \\ j_2 = & +0 & 000 & 000 & 000 \\ d_6 = & \text{ADD} & 0 & A^* & A_1 - 1 \end{array}$$

the flow diagram for modifying the orders of boxes 5.0 and 6.0 is constructed and is given in Fig. 3-3-9. In each of the words d_1, d_3, d_4, and d_5, the address $A_1 - 1$ appears. This is necessary since j is incremented before the calculations are performed. For example, when $j = 1$, boxes 1.0 through 1.5 produce the address $(A_1 - 1) + 1 = A_1$ for the commands of boxes 3.0, 4.0, 5.0, and 6.0.

By using another word n', located at N', where

$$n' = +0 \quad n_1' n_2' n_3' \quad 000 \quad 000$$

and n_1', n_2', and n_3' are the digits of $n + 1$, the computer flow diagram for box 2 of Fig. 3-3-4 may be constructed and is given in Fig. 3-3-10. Note that $j_1 \geq n'$ will be satisfied whenever $j > n$ since $j_1 = 10^{-4}j$ and $n' = 10^{-4}(n + 1)$. The comparison is made as shown in Fig. 3-3-10

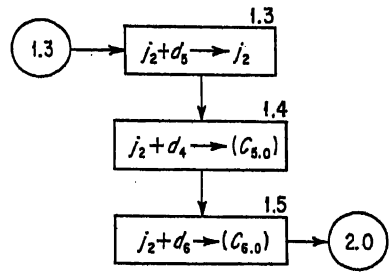


FIG. 3-3-9. Computer flow diagram for modifying the commands of boxes 5.0 and 6.0.

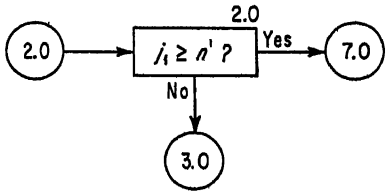


FIG. 3-3-10. Computer flow diagram for box 2 of Fig. 3-3-4.

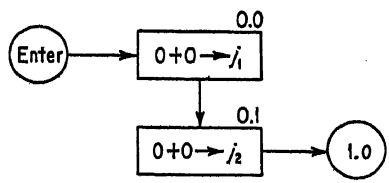


FIG. 3-3-11. Computer flow diagram for box 0 of Fig. 3-3-4.

since the three-address computer does not have a test for "greater than" but does for "greater than or equal to." The computer flow diagram for box 0 of Fig. 3-3-4 is given in Fig. 3-3-11. In boxes 0.0 and 0.1, the

counters j_1 and j_2, used for modifying the addresses A_j with subscript j, are set to zero.

The computer flow diagram for the three-address computer corresponding to the problem flow diagram of Fig. 3-3-4 is obtained by combining

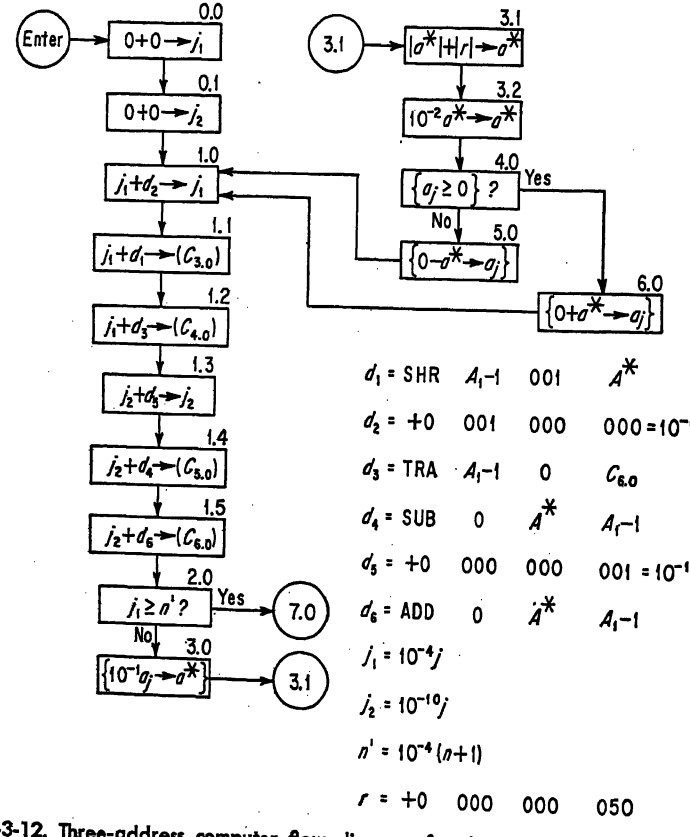


FIG. 3-3-12. Three-address computer flow diagram for the problem flow diagram of Fig. 3-3-4.

Figs. 3-3-5 through 3-3-11 and is given in Fig. 3-3-12. In this flow diagram, a list of constants and variables introduced solely for its construction appear so that the flow diagram may be easily understood. Also, we point out again that the commands for boxes 3.0, 4.0, 5.0, and 6.0 are modified by constructing them in the computer as shown in boxes 1.0 through 1.5. In Chap. 1, similar commands were modified directly, and fewer computer operations were required. However, since the initial

settings of such commands are lost during computation, the method used there causes more operations by the person operating the computer should it be desired to restart the problem, as is often required in the debugging (testing) of a routine. In the computer flow diagram of Fig. 3-3-12, the initial settings for the commands of boxes 3.0, 4.0, 5.0, and 6.0 remain at memory locations D_1, D_2, D_3, and D_6, respectively. Thus the initial settings are not lost. The method of modifying commands given in this flow diagram contains the basic idea for the construction of B-boxes which is described in the next chapter. B-boxes (also called index registers) are used to automatically modify commands.

To complete the computer flow diagram for the summation problem of Eqs. (3-2-1), (3-3-1), and (3-2-3), one may also draw a computer flow diagram for the second part of the flow diagram of Fig. 3-2-3. The result of such an effort is given in Fig. 3-3-13. The purpose of box 7.1 is to set $\bar{s} = 0$ before the summation is performed so that the program may be restarted without accumulating

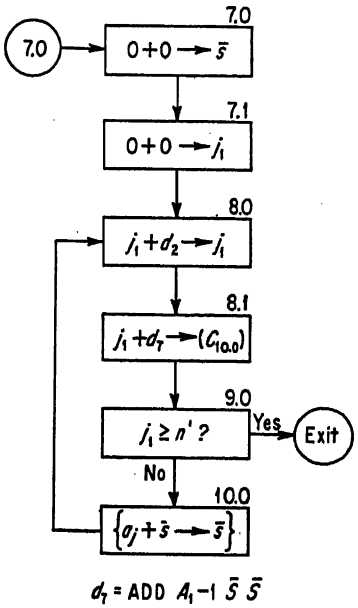


FIG. 3-3-13. Three-address computer flow diagram for second part of Fig. 3-2-3.

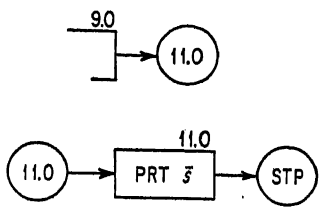


FIG. 3-3-14. Listing of answer for flow diagram of Figs. 3-3-12 and 3-3-13.

to previous sums. It should be further pointed out that Figs. 3-3-12 and 3-3-13 do not give sufficient information to run the problem on the computer since this flow diagram neither indicates how the code for the program is loaded into the computer nor how the answer is outputted after the computation is completed. (Later in this chapter we will discuss a routine for loading routines and data into the memory of the computer.) The answer $\bar{s}$ to this problem may be obtained by replacing the exit connector with the print command PRT 001 $\bar{S}$ M(STP) followed by STP (stop) order as shown in Fig. 3-3-14.

3-4. Three-address Computer Coding for $\bar{s} = \sum_{j=1}^{n} \bar{a}_j$

Using the flow diagrams of Figs. 3-3-12, 3-3-13, and 3-3-14 and the coding sheet of Table 3-4-1, we proceed to code the problem of Eqs. (3-2-1), (3-3-1), and (3-2-3) in much the same manner as in Chap. 1. That is, we first fill in the columns labeled Box, Order Symbol, and Branch. Then the assignment of memory addresses for each order, constant, variable, and datum is made. Again, as in Chap. 1, we will choose the addresses for orders to start at 100, for constants and variables to start at 200, and for the data a_j to start at 300 (assuming $A_n \leq 999$). The first 100 memory cells, addressed 000 through 099, are left open for special computer routines such as the load routine. It is often advantageous to reserve a block of memory locations for special computer routines. These memory locations are avoided by the programmer when coding a particular problem. In addition to the load routine, they are reserved for debugging routines, memory sum check routines, memory dump routines, etc. (routines which are used in conjunction with most problem routines).

There are two commands on the coding sheet which do not appear explicitly on the computer flow chart. These are the unconditional commands which follow the commands for boxes 5.0 and 6.0. These commands, however, are implied since the flow diagram indicates a reentry to a previous part of the program without a conditional transfer test. The words at memory locations 109, 112, 113, and 115 are enclosed in brackets to show that the computer constructs these orders. However, we fill in these words so that there will be the proper number of words in the routine and the load command of the three-address computer will properly load this routine.

The statement of the problem, the statement of the problem for the computer, the problem flow diagram, the computer flow diagram, and the coding represent a systematic method for constructing the computer code for a problem. We do not wish to imply that the code is constructed by such a step-by-step procedure, i.e., that the statement of the computer problem is generated from the original problem from scaling considerations only or that the problem flow diagram is drawn only with the aid of the statement of the computer problem, etc. In fact, the three-address code just constructed did not proceed in just this fashion. Equation (3-3-1) of the statement of the problem replaced Eq. (3-2-2) when considerations of the possible three-address commands were considered, and the first part of the problem flow diagram of Fig. 3-2-3 was redrawn in

TABLE 3-4-1. Coding for $\bar{s} = \sum_{j=1}^{n} \bar{a}_j$

Box	Order symbol				Branch	Memory location	Order code				Remarks
	I	M_1	M_2	M_3			I	M_1	M_2	M_3	
0.0	ADD	0	0	J_1	Enter	100	01	200	200	201	$j_1 = 10^{-4}j$
0.1	ADD	0	0	J_2		101	01	200	200	202	$j_2 = 10^{-10}j$
1.0	ADD	J_1	D_2	J_1		102	01	201	203	201	$d_2 = 10^{-4}$
1.1	ADD	J_1	D_1	$C_{2.0}$		103	01	201	204	109	
1.2	ADD	J_1	D_3	$C_{4.0}$		104	01	201	205	112	
1.3	ADD	J_2	D_5	J_2		105	01	202	206	202	$d_5 = 10^{-10}$
1.4	ADD	J_2	D_4	$C_{6.0}$		106	01	202	207	113	
1.5	ADD	J_2	D_6	$C_{8.0}$		107	01	202	208	115	
2.0	TRA	J_1	N'	$C_{7.0}$	7.0	108	22	201	209	117	$n' = 10^{-4}(n+1)$
3.0	{SHR	A_j	001	A^*}		109	{00	000	000	000}	
3.1	ADA	A^*	R	A^*		110	01	210	211	210	
3.2	SHR	A^*	002	A^*		111	25	210	002	210	
4.0	{TRA	A_j	0	$C_{8.0}$}	8.0	112	{00	000	000	000}	
5.0	{SUB	0	A^*	A_j}		113	{00	000	000	000}	A_j contains $\bar{a}_j$
	UCD	0	0	$C_{1.0}$	1.0	114	21	000	000	102	
6.0	{ADD	0	A^*	A_j}		115	{00	000	000	000}	A_j contains $\bar{a}_j$
	UCD	0	0	$C_{1.0}$	1.0	116	21	000	000	102	
7.0	ADD	0	0	$\bar{S}$		117	01	200	200	213	$\bar{s} = 10^{-8}s$
7.1	ADD	0	0	J_1		118	01	200	200	201	
8.0	ADD	J_1	D_8	J_1		119	01	201	203	201	
8.1	ADD	J_1	D_7	$C_{10.0}$		120	01	201	214	122	
9.0	TRA	J_1	N'	$C_{11.0}$	11.0	121	22	201	209	124	
10.0	{ADD	A_j	$\bar{S}$	$\bar{S}$}		122	{00	000	000	000}	
	UCD	0	0	$C_{8.0}$	8.0	123	21	000	000	119	
11.0	PRT	001	$\bar{S}$	M(STP)		124	42	001	213	125	
	STP	—	—	—		125	00	000	000	000	
	←	0	→			200	+0	000	000	000	
	←	j_1	→			201	+0	000	000	000	
	←	j_2	→			202	+0	000	000	000	
	$d_2 = 10^{-4}$					203	+0	001	000	000	
	$d_1 =$ SHR	$A_1 - 1$	001	A^*		204	25	299	001	210	
	$d_3 =$ TRA	$A_1 - 1$	0	$C_{4.0}$		205	22	299	200	115	
	$d_5 = 10^{-10}$					206	+0	000	000	001	
	$d_4 =$ SUB	0	A^*	$A_1 - 1$		207	02	200	210	299	
	$d_6 =$ ADD	0	A^*	$A_1 - 1$		208	01	200	210	299	
	←	n'	→			209	+0	000	000	$(n+1)$	
	←	a^*	→			210	+0	000	000	000	
	←	r	→			211	+0	000	000	050	
	←	$\bar{s}$	→			213	+0	000	000	000	
	$d_7 =$ ADD	$A_1 - 1$	$\bar{S}$	$\bar{S}$		214	01	299	213	213	
	←	a_1	→			300	←	a_1	→		
		⋮						⋮			
	←	a_n	→			300 + $(n-1)$	←	a_n	→		

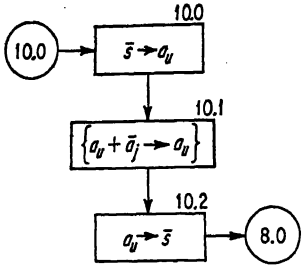


FIG. 3-5-1. Two-address computer flow diagram for box 10 of Fig. 3-2-3.

Fig. 3-3-4. Similarly, boxes 1.0 through 1.5 of the computer flow diagram were not constructed until we determined the coding for boxes 3.0, 4.0, 5.0, and 6.0. However, we do wish to imply that after the code is constructed it may be directly interpreted from the computer flow diagram, that the computer flow diagram may be directly interpreted from the problem flow diagram, etc. Thus, we have, at worst, devised an iterative scheme for producing a code which may be traced systematically to the statement of the problem. Furthermore, we do not wish to imply that this scheme will produce an error-free code. However, it does provide a complete picture so that one may check for coding errors.

3-5. Two-address Computer Flow Diagram

Again, let us consider the problem of Eqs. (3-2-1), (3-3-1), and (3-2-3) and construct a computer flow diagram for the second part of the flow diagram of Fig. 3-2-3. We use the two-address computer of Appendix II for the construction of this computer flow diagram. Further, we assume that the $\bar{a}_j$ have already been calculated and are stored in the sequential memory locations A_j where the a_j were originally located. The contents of the upper and lower accumulator are represented by a_U and a_L, respectively. For example, in a flow diagram, the expression $x \rightarrow a_L$ implies the command CAL X M_2 and the expression $a_U + y \rightarrow a_U$ implies the command HAU Y M_2 where in these examples M_2 is the address of the next order.

Starting with box 10 of Fig. 3-2-3, the two-address computer flow diagram is constructed as shown in Fig. 3-5-1. Again, as for the three-address computer flow diagram, braces are used to enclose commands containing variables, and each box contains

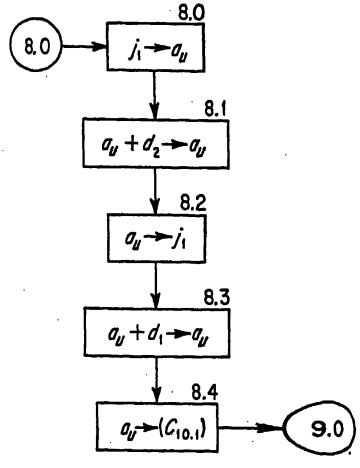


FIG. 3-5-2. Two-address computer flow diagram for box 8 of Fig. 3-2-3.

a single computer operation. For the computer flow diagram of box 8, two words are constructed. The first is

$$d_1 = \text{HAU} \quad A_1 - 1 \quad 0000$$

and the second is

$$d_2 = +00\ 0001\ 0000$$

Also, we will assume $j_1 = 10^{-6}j$. The computer flow diagram for box 8 is given in Fig. 3-5-2. Since, in box 8.2, the storing of the contents of A_U does not destroy a_U, d_1 is added to a_U in box 8.3 to create the command for box 10.1. The computer

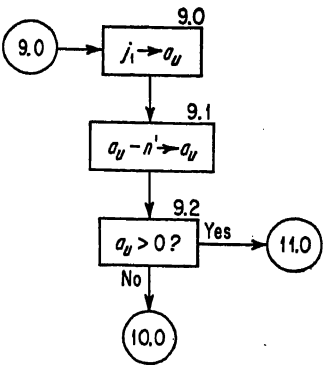


FIG. 3-5-3. Two-address computer flow diagram for box 9 of Fig. 3-2-3.

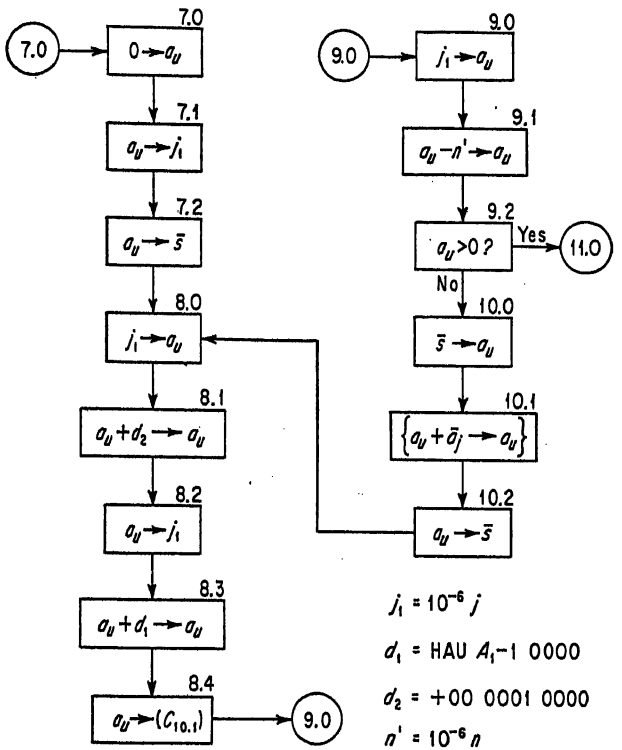


FIG. 3-5-4. Two-address computer flow diagram for second part of Fig. 3-2-3.

flow diagram for box 9 is to use the number $n' = 10^{-6}n$, i.e.,

$$n' = +00\ (0n_1'n_2'n_3')\ 0000$$

where $n_1'n_2'n_3'$ are the digits of n; the diagram is given in Fig. 3-5-3. In boxes 9.0 through 9.2, a_U designates the contents of the upper accumulator, and a test is performed which is equivalent to that of determining when j is greater than n. Even though the memory of the two-address computer may contain more than 1,000 words, n has at most three significant digits since the computer statement of the problem was posed and scaled in this fashion. The complete computer flow diagram for the second part of the flow diagram of Fig. 3-2-3, boxes 7 through 10, is given in Fig. 3-5-4.

TABLE 3-5-1. Two-address Coding for Flow Diagram of Fig. 3-5-4

Box	Order symbol			Branch	Memory location
	I	M_1	M_2		
7.0	CAU	0	0000		M
7.1	STU	J_1	0000		$M+1$
7.2	STU	$\bar{S}$	0000		$M+2$
8.0	CAU	J_1	0000		$M+3$
8.1	HAU	D_2	0000		$M+4$
8.2	STU	J_1	0000		$M+5$
8.3	HAU	D_1	0000		$M+6$
8.4	STU	$M+12$	0000		$M+7$
9.0	CAU	J_1	0000		$M+8$
9.1	HSU	N'	0000		$M+9$
9.2	TAP	$C_{11.0}$	0000	11.0	$M+10$
10.0	CAU	$\bar{S}$	0000		$M+11$
10.1	{HAU	A_j	000}		$M+12$
10.2	STU	$\bar{S}$	$M+3$	8.0	$M+13$

By letting M be the address of the order of box 7.0, the coding for the columns entitled Box, Order Symbol, Branch, and Memory Location is given for Fig. 3-5-4 in Table 3-5-1.

3-6. Detailed Flow Diagram

In learning to program and code, the construction of both the problem and the computer flow diagram is helpful and educational for the programmer. It is helpful because it forces the programmer to think through the problem, giving him a mental picture of possible arrangements for the

orders, variables, constants, and data before he codes the problem. It is educational since the construction of one part of a computer flow diagram is often dependent on the construction of other parts of the flow diagram and this interdependence reveals itself in tying the parts of the flow diagram together. For example, in the three-address computer flow diagram of Fig. 3-3-12, the construction of boxes 1.0 through 1.5 depends on boxes 3.0, 4.0, 5.0, and 6.0. However, as the programmer gains experience he will find a computer flow diagram more detailed than is necessary for coding. For example, boxes 10.0, 10.1, and 10.2 of the two-address computer flow diagram might just as well be replaced by the box 10.0 of the three-address flow diagram where it is assumed that this box requires three two-address orders for its code. Other ways of shortening the computer flow diagram become apparent as the programmer progresses.

As indicated in Sec. 1-5, the programmer (or coder) often paper-checks the coding of a problem or parts thereof. One such check is the construction of a dynamic memory chart. A dynamic memory chart for the coding of Table 3-5-1 is given in Table 3-5-2. In this table, the columns indicate the contents of important memory cells and the upper and lower accumulator at the end of each computer operation. Only the changes in the contents are recorded so that the table may be easily read. The computer operations are indicated by the box number corresponding to that of the coding sheet. The first row entry of the table indicates the contents of the memory cells and the accumulator at the start of the routine. Dashes are used to indicate that the contents are not initially important to the routine.

Since one of the tasks of a programmer may be to determine the appropriate computer for the solution of a given problem, he will often have to analyze the problem through the flow diagram stage before he can make the decision. He would then code parts of the problem for each of the computers so that he could make comparisons. Thus, he prefers to construct a single flow diagram which will serve him while coding for each computer.

A *detailed flow diagram* is a combination of a problem flow diagram and a computer flow diagram in that it describes the problem and makes use of the basic properties of most automatic internally coded digital computers. The contents of any box of a detailed flow diagram should not contain operations which are order dependent. If two operations are included in a box, then either can be done first. For example, if the variables j and s are set to zero in a single box, then the solution of the problem will not be changed by setting one to zero before the other. However, we can see that placing the two operations $0 \rightarrow j$ and $j + 1 \rightarrow j$

in the same box does not possess this property since whichever one is performed last sets the value of j. Since one could obviously devise a rule so that the ordering would be known in such cases, the reason for not allowing two such operations has another root for its existence. The reentry of a flow diagram to a box containing ordered operations would cause confusion, especially if not all operations were to be performed when the box was reentered.

TABLE 3-5-2. Dynamic Memory Chart for Coding of Table 3-5-1

Box	Address								Av	Ai
	0	J_1	$\bar{S}$	D_1	D_2	$M+12$	N'	A_j		
	0	—	—	d_1	d_2	—	n'	a_j	—	—
7.0	...	.....	.....	...	...	.....	...	...	0	0
7.1	...	0								
7.2	...	.....	0							
8.0	...	.....	.....	...	...	.....	...	...	$j_1 = 0$	
8.1	...	.....	.....	...	...	.....	...	...	$d_2 = 10^{-6}$	
8.2	...	10^{-6}								
8.3	...	.....	.....	...	...	.....	...	...	$d_2 + d_1 =$ HAU A_1 0	
8.4	...	.....	.....	...	...	HAU A_1 0				
9.0	...	.....	.....	...	...	.....	...	...	$j_1 = 10^{-6}$	
9.1	...	.....	.....	...	...	.....	...	...	$j_1 - n'$	
9.2	...	.....	.....	...	...	.....	...	...	<0 (if $n' > 1$)	
10.0	...	.....	.....	...	...	.....	...	...	$\bar{s} = 0$	
10.1	...	.....	.....	...	...	.....	...	...	$\bar{s} + a_1$	
10.2	...	.....	a_1							
8.0	...	.....	.....	...	...	.....	...	...	$j_1 = 10^{-6}$	
8.1	...	.....	.....	...	...	.....	...	...	$d_2 + 10^{-6} = 2{\cdot}10^{-6}$	
8.2	...	$2{\cdot}10^{-6}$								
8.3	...	.....	.....	...	...	.....	...	...	$2{\cdot}10^{-6} + d_1 =$ HAU A_2 0	
8.4	...	.....	.....	...	...	HAU A_2 0				
9.0	...	.....	.....	...	...	.....	...	...	$j_1 = 2{\cdot}10^{-6}$	
9.1	...	.....	.....	...	...	.....	...	...	$2{\cdot}10^{-6} - n'$	
9.2	...	.....	.....	...	...	.....	...	...	<0 (if $n' > 2$)	
10.0	...	.....	.....	...	...	.....	...	...	$\bar{s} = a_1$	
10.1	...	.....	.....	...	...	.....	...	...	$a_1 + a_2$	
10.2	...	.....	$a_1 + a_2$							
8.0	etc.									

The detailed flow chart for the problem of Eqs. (3-2-1), (3-3-1), and (3-2-3) which replaces the problem flow diagrams of Fig. 3-3-4 and the second part of Fig. 3-2-3 and either of the corresponding problem flow diagrams is given in Fig. 3-6-1. Boxes 1.0 and 8.0 imply not only that j is incremented, but also that commands within following boxes containing variables with subscript j are modified. Box 1.0 is used in modifying the addresses of a_j in boxes 3.0, 4.0, 5.0, and 6.0, and box 8.0 is used in modifying the address of a_j in box 10.0. Had the command of

box 10.0 been modified by box 1.0, the address of that command would have been corrected by the operations of box 8.0; and, therefore, it is dropped from the list of those modified by box 1.0. The operations of boxes 3.0, 4.0, 5.0, 6.0, and 10.0 are enclosed in braces to help the programmer locate the commands to be modified when he codes the problem. We point out that if the contents of boxes 3.1 and 3.2 were replaced by $10^{-2}(a^* + r) \rightarrow a^*$, the coding might be done as either $a^* + r \rightarrow a^*$ followed by $10^{-2}a^* \rightarrow a^*$ or as $10^{-2}a^* \rightarrow a^*$ followed by $10^{-2}r \rightarrow r^*$ and $a^* + r^* \rightarrow a^*$. The second method would not accomplish the desired result since $10^{-2}r \rightarrow r^*$ produces zero for r^* in the computer. However the stating of a rule that quantities within a parenthesis are computed separately would avoid this error. Since rules of this type are employed in the automatic coding schemes, we will not adopt them at this stage. We only mention that by generalizing this rule to operations involving expressions containing parentheses, such operations must have an even number of parentheses and that coding may be performed by sequentially removing the innermost pair. Then, by having a computer which can recognize a rather small set of special symbols, a computer could be designed or programmed that would write its own program for many problems. For example, a routine for evaluation of the expression

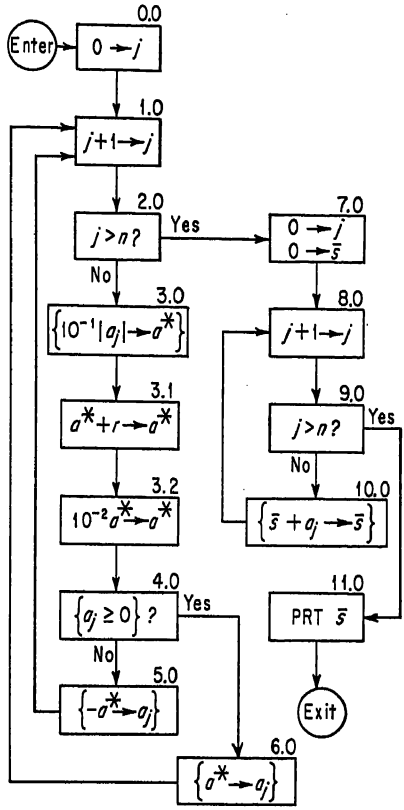


FIG. 3-6-1. Detailed flow diagram for $\bar{s} = \sum_{j=1}^{n} \bar{a}_j$.

$$a^* = 10^{-2}((10^{-1}(|a_j|)) + r)$$

might be coded by the computer.[1]

[1] Recently developed programs used for generating computer code from algebraic symbols use more sophisticated translation schemes. These programs are called compilers. Compilers are discussed in Chap. 8.

In the remainder of this book we will use detailed flow diagrams for descriptions of problems and use computer flow diagrams only to extend the explanations for parts of a detailed flow diagram. However, we recommend that the beginner continue to construct computer flow diagrams to aid in his learning.

3-7. A Load Routine

The load routine discussed in this section is assumed to be in a fixed location in the memory and is designed to load blocks of words, each block into sequential memory cells, either from instructions at the computer console or from instructions in a problem routine. The load routine, when directed to load a block of information, makes use of a *header word.* A header word contains information concerning the block of words to be loaded. It may be supplied to the load routine by a problem routine or it may immediately precede the block of words to be loaded. To describe the load routine of this section, we will use the three-address computer of Appendix I. The header word will have the form

$$h = 00 \quad M_1 \quad M_2 \quad N$$

where M_1 = number of words in block to be loaded
M_2 = first of sequential memory addresses assigned to block of words
N = special instruction for load routine

If N is 000 the routine will, after loading the designated block of words, read another header word from the input unit and the block of words associated with it. If N is nonzero, the computer will, after the designated block of words has been read, take its next instruction from memory location N.

In the flow diagram of Fig. 3-7-1, the notation $\text{XTR}(m_1,m_2) \rightarrow m_3$ represents the three-address XTR command of Appendix I. Similarly, LOD (M_1,M_2), M_3 represents the three-address LOD of Appendix I. By entering the routine at P, the load routine, in box 0, reads one word h from the input unit and stores it at H. In box 1, the routine makes sure that the digit positions s, a_0, a_1, . . . , a_6 of the test word a to be constructed are zero; and in box 2 the last three digits of the header word h, which are N, are extracted into a. In box 3, the routine determines which action the computer will follow after the routine (in box 5) has loaded the designated block of words. Note that the connector X is a variable connector.

As an example of the use of the preceding routine, where we assume

the load routine is already in the computer, let us consider the loading of the routine in Table 3-4-1. We construct three header words

$$h_1 = 00 \quad 026 \quad 100 \quad 000$$
$$h_2 = 00 \quad 015 \quad 200 \quad 000$$

and

$$h_3 = 00 \quad n \quad 300 \quad 100$$

The arrangement for the words to be read into the input unit is h_1 followed by order words to be read into memory cells addressed 100 through 125, h_2 followed by the constants and variables to be read into memory cells addressed 200 through 214, and h_3 followed by the data to be read into memory cells addressed 300 through 299 + n. To load, we enter P from the console into CC (the control counter) and press the computer start button. In this manner the computer starts the load routine at P, loads h_1 in location H which instructs the computer to load the next 26 words into memory locations 100 through 125 and returns the computer to P in the load routine. The routine then loads h_2 in location H which instructs the computer to load the next 15 words into memory locations 200 through 214 and again returns the computer to P in the load routine. Next, the routine loads h_3 in location H which instructs the computer to load the last n words into memory locations 300 through 299 + n and informs the computer to take its next instruction from memory location 100 which is the first order in the routine for $\bar{s} = \sum_{j=1}^{n} \bar{a}_j$.

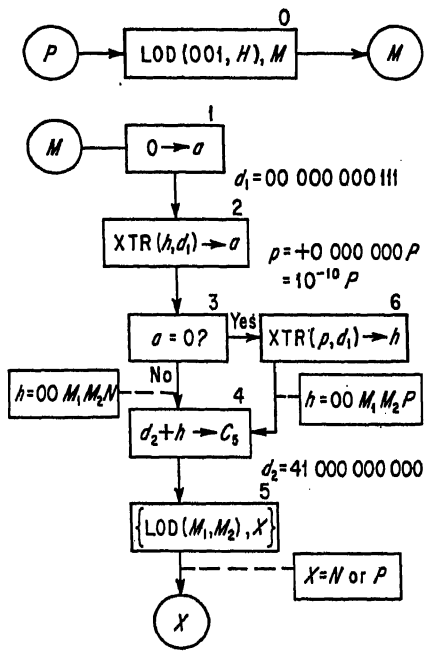


FIG. 3-7-1. Flow diagram for a load routine.

If in the previous example the load routine were not already located in the memory, then the 14 words of the load routine (assumed addressed 011 through 024) are placed in front of h_1. These 14 words are the eight order words for boxes 0 through 6 and the UCD command from box 6 to box 4 and the six variables and constants h, a, 0, d_1, d_2, and p. The

computer mode of operation switch is switched from "continuous" to "single operation" which permits the computer to perform only one operation when the start button is pushed. The command SWL M_1 —, where M_1 designates the input unit to be used, is placed in CR (the control register) and the start button pushed. The result of this action is to connect the desired input unit to the computer. Next, the order LOD 014 011 011 is placed in CR, the computer mode of operation switch is placed in the "continuous" position, and the start button is pushed. These actions cause the computer to load the load routine into memory locations 011 through 024 and then to enter the load routine at $P = 011$ which, in turn, causes the problem routine to be loaded as before.

Now, suppose we wish to use the routine for $\bar{s} = \sum_{j=1}^{n} \bar{a}_j$ to calculate several such sums. That is, we wish to calculate a set of $\bar{s}_k$ where $\bar{s}_k = \sum_{j=1}^{n_k} \bar{a}_{kj}$. This is accomplished by replacing the STP order addressed 125 in Table 3-4-1 by UCD 000 000 011, and instead of h_3 and the data cards now existing in the input words, we have sequences of data in the form h_{k1}, n'_k, h_{k2}, a_{kj}'s where

$$h_{k1} = 00 \quad 001 \quad 209 \quad 000$$
$$n'_k = +0 \quad 000 \quad 000 \quad (n_k + 1)$$

and

$$h_{k2} = 00 \quad n_k \quad 300 \quad 100$$

These sequences are followed by two words h' and t where

$$h' = 00 \quad 001 \quad 126 \quad 126$$

and

$$t = 00 \quad 000 \quad 000 \quad 000$$

which cause the computer to stop.

If, in a problem routine, it is desired to use the load routine to load n words into memory cells addressed R through $R + n - 1$ and return to the problem routine at location T, then the problem routine would construct the word

$$h = 00 \quad n \quad R \quad T$$

load it into location H of the load routine, and transfer control to (sets CC to) M.

3-8. Detailed Flow Diagram for $x = \sqrt{y}$

In Chap. 1 we considered the problem of obtaining $x = \sqrt{y}$ without considering whether the given word y and its square root x are computer

words, or if any of the operations in obtaining x cause a computer overflow. Let us now consider the problem of finding the square roots x_j for a given set of $y_j > 0$, where $j = 1, 2, \ldots, n \leq 400$, using fixed point computer operations. Since the magnitude of the largest word the computer will recognize is $1 - 10^{-10}$ and the square of this number is $1 - 2 \cdot 10^{-10} + 10^{-20}$ (which the computer will recognize as $1 - 2 \cdot 10^{-10}$), we will restrict the y_j to those which satisfy

$$y_j \leq 1 - 2 \cdot 10^{-10}$$

Also, since the square of 10^{-5} is 10^{-10}, we will assume $y_j = 0$ if $0 \leq y_j < 10^{-10}$. Thus, the range for y_j is

$$10^{-10} \leq y_j \leq 1 - 2 \cdot 10^{-10} \tag{3-8-1}$$

Furthermore, we will compute only the positive square roots $x_j > 0$. Had we desired $\xi_j = \sqrt{\eta_j}$ where η_j did not satisfy the inequality (3-8-1), then η_j would be multiplied by a scale factor ρ_j where ρ_j is so chosen that $y_j = \rho_j \eta_j$ satisfies (3-8-1). The desired answer would be

$$\xi_j = \sqrt{\eta_j} = \frac{\sqrt{y_j}}{\sqrt{\rho_j}} = \frac{x_j}{\sqrt{\rho_j}}$$

where the computer computes x_j. For example, suppose $10^2 \leq \eta_j < 10^8$, then by choosing $\rho_j = 10^{-4}$, $y_j \doteq 10^{-4}\eta_j$ will satisfy condition (3-8-1) and $\xi_j \doteq 10^2 x_j$.

The problem may now be stated as: find each $x_j = x > 0$, where $x = \sqrt{y}$, for the set $y = y_j$, $j = 1, 2, \ldots, n \leq 400$, by repeated use of the formula

$$w_{i+1} = \frac{1}{2}\left(\frac{y}{w_i} + w_i\right) \tag{3-8-2}$$

(Newton's method) where

$$w_0 = \frac{1}{2} + \frac{y}{2} \tag{3-8-3}$$

and the approximation for the square root is

$$x = w_{i+1} \tag{3-8-4}$$

for the smallest i such that

$$|w_{i+1} - w_i| \leq \epsilon \qquad \epsilon \geq 10^{-10} \tag{3-8-5}$$

The preceding is not a computer statement of the problem since we have not prescribed the order in which the operations of Eq. (3-8-2) are to be performed. To determine this order we look at possible operations

which may cause a computer overflow. It may be shown that if the operations are performed exactly

$$0 < y < \sqrt{y} \leq w_{i+1} \leq w_i < 1 \tag{3-8-4}$$

for $i = 0, 1, 2, \ldots$, when y satisfies the condition (3-8-1) for the y. If the initial approximation w_0 is determined by Eq. (3-8-3) and $y > \frac{1}{2}$, then

$$\frac{1}{2} < \frac{y}{w_i} < 1$$

and the quantity

$$\left(\frac{y}{w_i} + w_i\right)$$

of Eq. (3-8-2) will be greater than 1. Thus, the right side of Eq. (3-8-2) cannot be calculated by calculating the quantity within the parentheses first. If we rewrite this equation as

$$w_{i+1} = \frac{1}{2}\left(\frac{y}{w_i}\right) + \frac{1}{2} w_i \tag{3-8-5}$$

then, even though $0 < y < w_i < 1$ for exact representations of the quantities y and w_i, by using rounded operations an overflow may occur. Since $w_0 = 1 - 10^{-10}$ when $y = 1 - 2 \cdot 10^{-10}$, $y/w_0 = 1 - 10^{-10}$ and the rounded operation $0.5\,(1 - 10^{-10}) = 0.5$, and $w_1 = w_{i+1} = 1$. However, if we write Eq. (3-8-2) as

$$w_{i+1} = \frac{0.5y}{w_i} + 0.5w_i \tag{3-8-6}$$

(i.e., first multiply y by 0.5, divide by w_i, and then add $0.5w_i$), we will not get an overflow when using rounded operations. We note that Eq. (3-8-7) does not produce an overflow when unrounded operations are used and $\frac{1}{2}$ is used as the multiplicative constant 0.5.

We now state the computer problem as: find each $x_j = x > 0$, where $x = \sqrt{y}$, for the set $y = y_j$, where each y_j satisfies the condition (3-8-1) and $j = 1, 2, \ldots, n \leq 400$, by repeated use of the formula

$$w_{i+1} = \frac{(ay)}{w_i} + aw_i \tag{3-8-8}$$

where
$$a = +5\ \ 000\ \ 000\ \ 000 \tag{3-8-10}$$

$$w_0 = a + ay \tag{3-8-11}$$

and
$$x = w_{i+1} \tag{3-8-12}$$

when
$$z = |w_{i+1} - w_i| \leq \epsilon \qquad \epsilon \geq 10^{-10} \tag{3-8-13}$$

The detailed flow diagram for this problem is given in Fig. 3-8-1. It is assumed that y_j are stored in sequential memory locations, and the x_j when computed are also stored in sequential memory locations. In general, the flow diagram is self-explanatory. However, we point out that z, as calculated in box 9, is $|w_{i+1} - w_i|$ since $w_i \geq w_{i+1}$. Also, since i is not recorded and there are only the two locations W_{i+1} and W_i for w_{i+1} and w_i respectively, i is automatically incremented when, in box 11, w_i is replaced by w_{i+1}. Since z is formed in box 9 and not used after box 10 until reformed, the location Z might just as well be used for the location W' indicated in the flow diagram. Finally, a list of square roots x_j is not very useful unless one knows the numbers from which they are obtained; so boxes 16 and 17 are used to list the x_j and the y_j in pairs.

The load routine of the preceding section can be used for loading the routine of Fig. 3-8-1. Should n exceed 400, then the y_j could be divided into groups of less than 400, each group having header cards. Then, by having the exit connector of Fig. 3-8-1 go to the connector P in the load routine, the x_j may be calculated in groups of 400 or less.

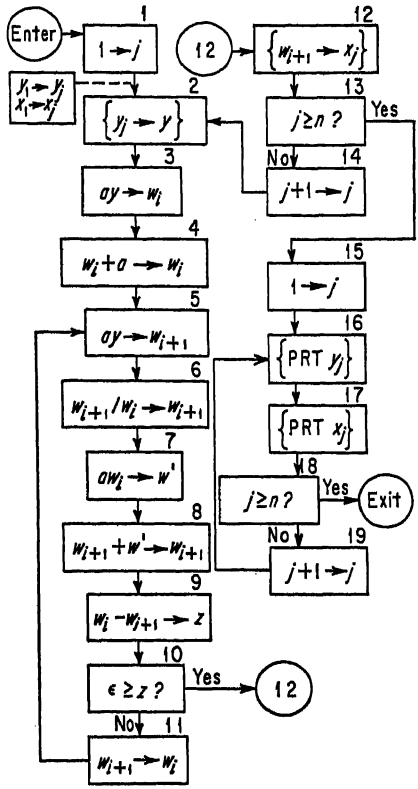


FIG. 3-8-1. Detailed flow diagram for $x_j = \sqrt{y_j}$ by Newton's method. Note: Boxes 16 and 17 may be included between boxes 12 and 13 thereby eliminating boxes 15, 18, and 19. The present flow diagram, with a slight modification, would allow printing more than one set of values Y_j and X_j. This may increase the speed of printing.

Another way of calculating $x = \sqrt{y}$ by Newton's method is to calculate $\Delta w_i = w_i - w_{i+1}$ where the equation for Δw_i is obtained by subtracting w_{i+1} as defined in Eq. (3-8-2) from w_i, i.e.,

$$\Delta w_i = w_i - w_{i+1} = w_i - \frac{1}{2}\left(\frac{y}{w_i} + w_i\right) = \frac{1}{2}\left(w_i - \frac{y}{w_i}\right) \quad (3\text{-}8\text{-}14)$$

The advantage of this equation is that the operations within the bracket may be performed first, thereby reducing the number of multiplications in the square root routine. The square root is given by

$$x = w_0 - \sum_{i=0}^{I} \Delta w_i \tag{3-8-15}$$

where I is the smallest value of i for which

$$\Delta w_i \leq \epsilon$$

3-9. Binary, Octal, and Hexadecimal Computers

From the point of view of computer design, the binary computer is the easiest type to construct since the base 2 contains only the two digits 0 and 1. The operations of the arithmetic unit, the complementing of numbers, and the storing of information is greatly simplified electronically by the use of such a simple number base. A 10-digit decimal word allows the representation of 10^{10} different magnitudes for numbers. The equivalent word size for numbers represented by a binary base is obtained from the equation $2^y = 10^{10}$ and the word size is $y = 10/\log 2$ or approximately 33.219 bits (binary digits). However, since the base-10 digits are usually represented in a high-speed digital computer by some form of binary coding and the simplest such code is the *binary-coded decimal digit* where each decimal digit is represented by four bits, i.e.,

$$\text{Decimal digit} \begin{cases} 0 = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 0000 \\ 1 = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0001 \\ 2 = 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 0010 \\ 3 = 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 0011 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ 9 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1001 \end{cases} \text{binary code}$$

a 10-decimal digit word may be considered as equivalent to 40 inefficiently used bits. That is, four bits may be used to represent 16 different configurations of which only ten are used in a base-10 computer. In actuality, most binary computers in use have word sizes which vary between 28 and 48 bits. For order words, the binary computers are usually considered to fall into two general classes: (1) the single-address command with two commands per order word, and (2) the one-, two-, or three-address command with a single command per order word. Before discussing these types of computers, let us consider the means of getting information in and out of them.

Since a binary word containing in the neighborhood of 40 bits consumes considerable space on a written page, is hard to commit to memory, and is tedious to write, it is desirable to shorten its representation. This is commonly done by giving its equivalent octal or hexadecimal number representation. Let e be a number of magnitude less than 1 and

$$e = \sum_{i=1}^{n} e_i 8^{-i} \tag{3-9-1}$$

where the e_i are integers such that $0 \leq e_i \leq 7$, be the octal representation of the magnitude of a number (here i runs from 1 to n instead of 0 to $n - 1$, so that simpler expressions are used in our explanation). Let

$$e = \sum_{j=1}^{3n} b_j 2^{-j} \tag{3-9-2}$$

where the b_j are either the bit 1 or 0, be the binary representation of the same magnitude. Then, equating Eqs. (3-9-1) and (3-9-2) gives

$$\begin{aligned}\sum_{i=1}^{n} e_i 8^{-i} = \sum_{i=1}^{n} e_i 2^{-3i} &= \sum_{j=1}^{3n} b_j 2^{-j} \\ &= \sum_{i=1}^{n} b_{3i-2} 2^{-3i+2} + \sum_{i=1}^{n} b_{3i-1} 2^{-3i+1} + \sum_{i=1}^{n} b_{3i} e^{-3i} \\ &= \sum_{i=1}^{n} (b_{3i-2} 2^2 + b_{3i-1} 2^1 + b_{3i} 2^0) 2^{-3i}\end{aligned}$$

Thus, by equating coefficients, one obtains

$$e_i = b_{3i-2} 2^2 + b_{3i-1} 2^1 + b_{3i} 2^0 \tag{3-9-3}$$

Since the coefficients b_{3i-2}, b_{3i-1}, and b_{3i} are either 1 or 0, the right-hand side of Eq. (3-9-3) can assume only one of the values 0, 1, 2, 3, 4, 5, 6, or 7, which are exactly the digits of the base-8 number representation. Thus, grouping the binary bits of a binary number representation in consecutive groups of three, each group can be assigned its equivalent base-8 representation. For example, the binary number

$$.101010101 = \tfrac{1}{2} + \tfrac{1}{8} + \tfrac{1}{32} + \tfrac{1}{128} + \tfrac{1}{512}$$

when grouped as .101 010 101 has the base-8 representation

$$.525 = \tfrac{5}{8} + \tfrac{2}{64} + \tfrac{5}{512}$$

However, $\tfrac{5}{8} = \tfrac{1}{2} + \tfrac{1}{8}$, $\tfrac{2}{64} = \tfrac{1}{32}$, and $\tfrac{5}{512} = \tfrac{1}{128} + \tfrac{1}{512}$: and the two representations .101010101 in binary, and .525 in octal, are different representations of the same magnitude. Thus, if the binary-word size is

$3n$ bits, by a simple code (Table 2-2-1), the input and output of the computer may be in the form of words with n digits in octal representation.

Similarly, let h be the base-16 representation of the magnitude of a number; i.e.,

$$h = \sum_{i=1}^{n} h_i(16)^{-i} \tag{3-9-4}$$

where each of the h_i are one of the base-16 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f; and let

$$h = \sum_{j=1}^{4n} b_j 2^{-j} \tag{3-9-5}$$

where the b_j are either 0 or 1, be the binary representation of the same magnitude. Then from Eqs. (3-9-4) and (3-9-5) one obtains

$$h_i = b_{4i-3}2^3 + b_{4i-2}2^2 + b_{4i-1}2^1 + b_{4i}2^0 \tag{3-9-6}$$

and the right-hand side of Eq. (3-9-6) may be used as a binary code for the 16 digits of the hexadecimal representation. Grouping the bits of a binary representation in consecutive groups of four, each group can be assigned its hexadecimal equivalent. For example, the hexadecimal equivalent of .1110 0100 1010 1000 is .$e4a8$. Thus, if the binary word size is $4n$ bits, the input and output of the computer may be in hexadecimal representation.

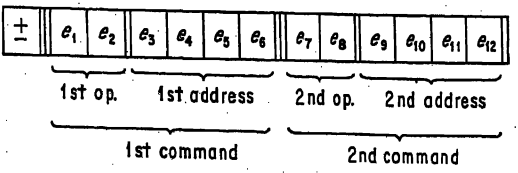


FIG. 3-9-1. Octal order word with two single-address commands.

The preceding discussion implies that the principal difference between the binary, octal, and hexadecimal computers is in the representation of the words as they pass into or out of the input or output units. The arithmetic unit and internal memory unit of each type of computer treats the information in the binary representation. If the word length is 36 bits and a sign digit, an order word may contain two commands, each having an operand and an address. The number representation used by the programmer is octal with two octal digits for the operation and four octal digits for the address. The configuration of such an order word is shown in Fig. 3-9-1. The normal operation of a computer with

such an order word format is to perform the first command of a word; next, the second command of the same word; then the first command of the next word, etc. A data word contains a sign digit and 12 octal digits. The codes and input information are usually compiled in octal representation; however, routines are constructed for such computers which allow them to accept binary-coded decimal information and convert it to equivalent binary information. Since the address of such a computer command contains four octal digits, the internal memory may be as large as $8^4 = 4{,}096$ words.

If the word length of the binary computer is 40 bits, each order word may contain two single-address commands. However, in this case, the

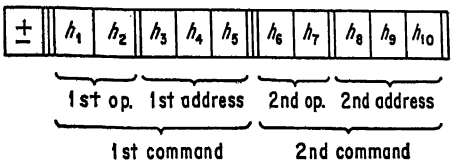


FIG. 3-9-2. Hexadecimal order word containing two single-address commands.

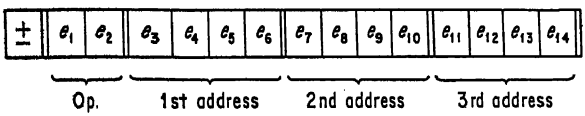


FIG. 3-9-3. Three-address order for an octal computer.

commands are represented by hexadecimal digits: the operand containing two hexadecimal digits and the address containing three hexadecimal digits. The configuration for such an order word is shown in Fig. 3-9-2. Since such a command has three hexadecimal digits for the address, the internal memory may be as large as $16^3 = 4{,}096$ words.

The preceding two-order word configurations classify the computer as being "single-address with two commands per order word." Should the memory be larger than 4,096 words, the format of the order words would have to be changed and a possible configuration would be the two-address computer similar to the two-address decimal computer already discussed.

If the word length of the binary computer is 42 bits, the order word could be of the form of a three-address computer with octal representation of numbers and commands. The word length would be 14 octal digits, allowing two digits for the operation and four digits for each of the three addresses as shown in Fig. 3-9-3. Again, such a computer may have an internal memory as large as 4,096 words. Similarly, if the word

size of the binary computer is 48 binary bits, the order word could be of the form of a three-address computer with hexadecimal representation. In such a case, the operand could be represented by two hexadecimal digits and each of the three addresses by three hexadecimal digits. The extra hexadecimal digit might be used as part of the operand or might be used as a special indicator such as indicating that one or more of the addresses refers to an auxiliary memory instead of the main memory, that the order word is to be treated as two single-address commands instead of a single three-address command, that the order is altered by a *B*-box, or that the command implies a floating point operation instead of the normal arithmetic operation.

Thus, except for converting the data from decimal to octal or hexadecimal, which may be done by a special routine within the computer, or done manually before being put in the computer, the methods of programming and coding a binary computer possess no major departures from those of the decimal computer.[1]

PROBLEMS

3-1. Construct a two-address computer flow diagram for the problem flow diagram of Fig. 3-3-4.

3-2. Using the result of Prob. 3-1 and Fig. 3-5-4, construct a two-address code for

$$\bar{s} = \sum_{j=1}^{n} \bar{a}_j$$

3-3. Construct a code for a two-address load routine.

3-4. Construct a three-address computer flow diagram and code for the flow diagram of Fig. 3-8-1.

3-5. Construct a two-address computer flow diagram and code for the flow diagram of Fig. 3-8-1.

3-6. A routine check sum is a number which may be used as a partial check for the loading of a routine. It is acquired by adding together all commands and constants of the routine. Since this process usually causes overflows, the overflows are neglected. Construct a routine which will sum all the words between two designated locations in the memory (neglecting overflows), compare this sum with a given number, and either transfer computer control to a designated location if sum and designated number are the same or list the sum and designated number and stop the computation if they are different.

[1] There are programming systems for most computers, called assembly systems, which are designed automatically to convert the orders expressed with mnemonic symbols for order symbols and addresses and decimal representations of constants and data into a corresponding code for the computer. These systems are described in Chap. 8.

4

CODING WITH B-BOXES

4-1. Introduction

A large portion of the programming and coding effort for a problem is expended in bookkeeping; that is, in keeping track of the amount and location of data already processed and that yet to be processed. Commands which reduce the amount of bookkeeping are often incorporated in computers. Commands associated with *B*-boxes, a British invention, fall into this category. *B*-boxes are used for address modification, for counting and for some other logical operations. Synonyms for the term *B*-box are *index register*, *step register*, and *address modification register*. In the next section we will program and code a simple problem and then use this problem in the description of the *B*-box commands.

4-2. The Sum of a Set of Numbers

Let us reconstruct the simple routine of Chaps. 1 and 3 which forms the sum s of a set of n numbers $a_1, a_2, \ldots, a_n$, i.e.,

$$s = \sum_{i=1}^{n} a_i \tag{4-2-1}$$

By letting

$$s_j = \sum_{i=1}^{j} a_i$$

for $1 \leq j \leq n$, and $s_0 = 0$, then we may write

$$s_j = a_j + s_{j-1}$$

and for $j = n$, $s = s_j$. The flow chart for this problem is given in Fig. 4-2-1. In constructing the flow chart it is assumed that n is a variable depending upon the number of terms, a_j, to be summed, except that n is

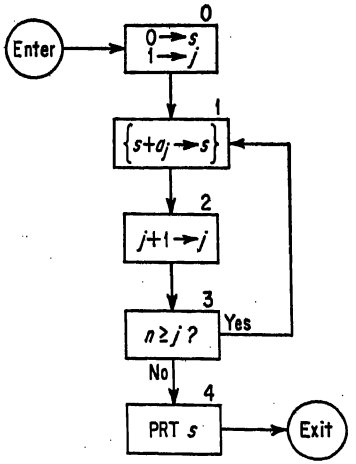


FIG. 4-2-1. Flow diagram for $s = \sum_{i=1}^{n} a_i$.

assumed greater than or equal to one, but does not exceed a number N determined by the availability of space in the main memory for the storage of the numbers a_j. Using the three-address code of Appendix I, a coding for this routine is given in Table 4-2-1. In coding the routine, it is assumed that the a_j are scaled such that $|a_j| < 1$ and $|s_j| < 1$ for all j, $1 \leq j \leq n$. It is further assumed that before entering the routine the a_j have been loaded into their proper memory cells and n has been stored in the memory cell 014 as $n' = 10^{-7}\, n$. The subscript j of s_j is dropped since only one memory cell is required for this variable, i.e., s_j is stored in the same cell where s_{j-1} was. In the coding of Table 4-2-1, the order

TABLE 4-2-1. Three-address Code for Flow Diagram of Fig. 4-2-1

Box no.	Order symbol. Operation	M_1	M_2	M_3	Branch	Memory location	Order code Operation	M_1	M_2	M_3
	←		0	→		000	+0	000	000	000
0.0	ADD	0	0	S	Enter	001	01	000	000	010
0.1	ADD	D_2	0	J'		002	01	013	000	011
0.2	ADD	D_1	0	$C_{1.0}$		003	01	012	000	004
1.0	{ADD	S	A_j	S}		004	{01	010	015	010}
2.0	ADD	J'	D_2	J'		005	01	011	013	011
2.1	ADD	$C_{1.0}$	J'	$C_{1.0}$		006	01	004	011	004
3.0	TRA	N'	J'	$C_{1.0}$	1.0	007	22	014	011	004
4.0	PRT	001	S	M(STP)		008	42	001	010	009
	STP	0	0	0		009	00	000	000	000
	←		s	→		010	←		s	→
	←		j'	→		011	←		$j' = 10^{-7}j$	→
	←		d_1	→		012	01	010	015	010
	←		d_2	→		013	+0	000	001	000
	←		n'	→		014	←		$n' = 10^{-7}n$	→
	←		a_1	→		015	←		a_1	→
	←		a_2	→		016	←		a_2	→
			.			.			.	
			.			.			.	
			.			.			.	
	←		a_n	→		n + 14	←		a_n	→

TABLE 4-2-2. Alternate Three-address Code for Flow Diagram of Fig. 4-2-1

Box no.	Order symbol: Operation	M_1	M_2	M_3	Branch	Memory location	Order code: Operation	M_1	M_2	M_3
	←		0	→		000	+0	000	000	000
0.0	ADD	0	0	S	Enter	001	01	000	000	009
0.1	ADD	N'	D_1	D_3		002	01	013	010	012
0.2	ADD	D_2	D_1	$C_{1.0}$		003	01	011	010	004
1.0	{ADD	S	A_j	S}		004	{01	009	014	009}
2.0	ADD	$C_{1.0}$	D_2	$C_{1.0}$		005	01	004	011	004
3.0	TRA	D_3	$C_{1.0}$	$C_{1.0}$	1.0	006	22	012	004	004
4.0	PRT	001	S	M(STP)		007	42	001	009	008
	STP	0	0	0		008	00	000	000	000
	←		s	→		009	←		s	→
	d_1 = ADD S A_0 S					010	01	009	013	009
	←		d_2	→		011	+0	000	001	000
	←		d_3	→		012	00	000	000	000
	←		n'	→		013	←		$n \times 10^{-7}$	→
	←		a_1	→		014	←		a_1	→
	←		a_2	→		015	←		a_2	→
			⋮						⋮	
	←		a_n	→		$n + 13$	←		a_n	→

0.2, the order ADD 0 D_1 $C_{1.0}$ is for presetting the variable second address in the command 1.0. The word d_1 is the order ADD S A_1 S. This allows the routine to be reused after the A_1 address has been modified. Thus, the routine may be reentered after a new set of a_j has been loaded in the memory without reloading the routine. The order 2.1, ADD $C_{1.0}$ J' $C_{1.0}$, changes the command 1.0 so that successive a_j are added to s. The value of j' is chosen to be $j' = 10^{-7}\, j$ and the address of a_j is modified after a_j is added to s. The braces around the order 1.0 indicate that the order is modified by the program. The command 2.0, ADD J' D_2 J', where $d_2 = 10^{-7}$, is used for the equivalent of incrementing j by 1. The order 3.0 asks if $n' \geq j'$, which is equivalent to asking if $n \geq j$. Similarly, the order 0.1, ADD D_2 0 J', initially sets j' to 10^{-7}, which is equivalent to initially setting j to one. The exit order for this routine is a stop command; however, such an exit command may be replaced by an unconditional transfer to another routine.

The computer running time for the preceding program may be

decreased by reducing the number of commands in box 2 of the flow diagram. This may be done by using the address of a_j in the order 1.0 as a counter instead of having a separate counter for j. The coding is shown in Table 4-2-2. The test of the order 3.0 is that of testing the command $C_{1.0}$,

$$\{\text{ADD } S\ A_j\ S\}$$

against the number $d_8 = \text{ADD } S\ A_n\ S$, where d_8 is created in the order 0.1. The operations for box 1 have been expanded to include the setting of the upper limit for the comparison test, *cf.* command $C_{0.1}$.

4-3. Expanded Word Structure for the Three-address Computer

It is assumed that the sign of each word is represented by a binary-coded hexadecimal digit. The operation code for order words consists of the two digits s and d_0 with the permissible values for s (for order words)

TABLE 4-3-1. Partial Table of the Operation Codes for Three-address Commands

Symbol	Operation code		Symbol	Operation code		Symbol	Operation code		Symbol	Operation code	
	s	d_0		s	d_0		s	d_0		s	d_0
Special			Logical			Operational			Floating point arithmetic		
STP	0	0	UCD	2	1	LOD	4	1	FAD	8	1
Arithmetic			TRA	2	2	PRT	4	2	FSU	8	2
ADD	0	1	OVW	2	3	SWL	4	3	FMR	8	3
SUB	0	2	XTR	2	4	SWP	4	4	FDR	8	5
MLR	0	3	SHR	2	5	BKP	4	5	FAA	8	7
MLT	0	4	SHL	2	6						
DVR	0	5									
DIV	0	6									
ABA	0	7									
SBA	0	8									

being the digits 0 through 9 (binary representations 0000 through 1001). For data words the binary representation 1010 is used for s to indicate that the word is a data word and is positive, and the binary representation 1011 is used for s to indicate that the data word is negative. In

this manner s may be used to distinguish between data words and order words. Table 4-3-1 gives the operation codes for the three-address orders of Appendix I except for the B-box commands introduced in this chapter. In this table, the orders are arranged in groups. The operational orders are separated from the logical orders since each operational order may require an action by the computer operator such as the loading of a deck of cards in an input or output unit, the setting of a console switch, etc. Except for the data being in floating point notation, the description of floating point commands (for which $s = 8$) follows the description of the corresponding fixed point equivalents in which $s = 0$. The operation codes having $s = 6$ are reserved for the magnetic-tape commands discussed in a later chapter, and those having odd digits for s will be discussed in the following section.

4-4. B-boxes (Index Registers)

A B-box is a register used in the modification of addresses during execution of an order without changing the order as stored in the memory. For example, in the problem of this chapter, the address for a_1 in the order 1.0 of Table 4-2-1 might be modified by a B-box to the successive addresses of the numbers a_j. B-boxes and B-box commands vary in design and operation from one computer to another. Here we will discuss a possible arrangement for the three-address computer of Appendix I. B-boxes for other computers operate in the same general manner but may differ in detail, particularly in the physical location of the registers and the method of tagging the address to be modified. Many computers have separate B-boxes and do not use the main memory for this purpose.

Except for the location 000, any main memory location of the three-address computer may be used as a B-box. Thus, the programmer may have as many B-boxes as required for a routine within the limits of available memory locations. Each B-box is designated by its memory location. Associated with the operation of the B-boxes are three orders: a SET command, an INC (increment and test) command, and a DEC (decrement and test) command. These orders are given in Table 4-4-1 where the contents b of a B-box is assumed to be a data word with digits designated as

$$+b_0b_1b_2b_3b_4b_5b_6b_7b_8b_9$$

In the INC and DEC commands, the terminology "N(mod 1,000)," which is read as N modulo one thousand, is used. The definition of this terminology is given in Chap. 6 on parallel and serial modes of computer

TABLE 4-4-1. Three-address B-box Commands

Symbol	Operation code		$M_1M_2M_3$	Explanation
	s	d_0		
SET	2	7	$M_1M_2M_3$	Set B-box with initial value M_1 and final value M_3 where the address of the B-box is $M_2 = d_4d_5d_6$. This order causes $b_i = d_i$ for $i = 1, 2, 3, 7, 8$, and 9, $b_0 = b_4 = b_5 = b_6 = 0$ and $s = +$, i.e., the contents of the B-box addressed M_2 is set equal to $+0\ M_1\ 000\ M_3$.
INC	2	8	$M_1M_2M_3$	Increment B-box (addressed $M_2 = d_4d_5d_6$) by the amount M_1. This causes $[b_1b_2b_3 + d_1d_2d_3] \pmod{1{,}000} \rightarrow b_1b_2b_3$. Next, test to see if $b_1b_2b_3 < b_7b_8b_9$. If $b_1b_2b_3 < b_7b_8b_9$, the next order is taken from memory location $M_3 = d_7d_8d_9$. If $b_1b_2b_3 \geq b_7b_8b_9$, the next order is taken sequentially. Note that if $b_7b_8b_9 = 000$, then the next command is taken in sequence.
DEC	2	9	$M_1M_2M_3$	Decrement B-box (addressed $M_2 = d_4d_5d_6$) by the amount M_1. This causes $[b_1b_2b_3 + (1{,}000 - d_1d_2d_3)] \pmod{1{,}000} \rightarrow b_1b_2b_3$. Next, test to see if $b_1b_2b_3 > b_7b_8b_9$. If $b_1b_2b_3 > b_7b_8b_9$, the next order is taken from memory location $M_3 = d_7d_8d_9$. If $b_1b_2b_3 \leq b_7b_8b_9$, the next order is taken sequentially. Note that if $b_7b_8b_9 = 999$, the next command is taken in sequence.

Note 1. M_1 consists of the digits d_1, d_2, and d_3; M_2 consists of the digits d_4, d_5, and d_6; and M_3 consists of the digits d_7, d_8, and d_9; the contents of B-box M_2 are $+b_0\ b_1b_2b_3\ b_4b_5b_6\ b_7b_8b_9$.

Note 2. Using an increment of 999 (the modulo equivalent of -1) would cause a downward count in the B-box. Similarly a decrement of 998 would cause an upward count of 2. Thus, the principal difference in the use of INC and DEC commands is in the branching condition.

operation. Here it suffices to say that if $0 \leq N < 1{,}000$, then N is not altered; but if $1{,}000 \leq N < 2{,}000$, then N is replaced by $N - 1{,}000$.

An order, an address of which is to be modified by a B-box, is tagged in the following way. The operation-code digit s is replaced by $s + 1$; that is, a B-boxed fixed point arithmetic order has $s = 1$, a B-boxed logical order has $s = 3$, a B-boxed operational order has $s = 5$, a B-boxed magnetic-tape order has $s = 7$, and a B-boxed floating point arithmetic order has $s = 9$. Whenever s is an odd number in an order word the

computer considers the next sequential word in memory as a part of that order. Let L be the address of the B-boxed order

$$\underbrace{sd_0}_{\text{op}} \quad \underbrace{d_1d_2d_3}_{M_1} \quad \underbrace{d_4d_5d_6}_{M_2} \quad \underbrace{d_7d_8d_9}_{M_3}$$

where s is an odd digit, then the word

$$sd_0 \quad \underbrace{d_1d_2d_3}_{B_{M_1}} \quad \underbrace{d_4d_5d_6}_{B_{M_2}} \quad \underbrace{d_7d_8d_9}_{B_{M_3}}$$

with address $L + 1$, gives the following interpretation to the order addressed L. The address M_1 is modified by the B-box addressed B_{M_1}, the address M_2 is modified by the B-box addressed B_{M_2}, and the address M_3 is modified by the B-box addressed B_{M_3}. Each of these addresses is modified when the order is transmitted from memory to the control register by adding the number $b_1b_2b_3$ of the designated B-box to the address and placing the sum in its proper address location in the control register. Should the modified address exceed 999, then the modified address (mod 1,000) is placed in the control register, that is, the overflow digit is dropped. Thus, the control register contains the order with modified addresses and the memory retains the order with unmodified addresses. Since the command in memory is unchanged, the address does not need to be reset to its original value for subsequent use. By having the memory location 000 contain +0 000 000 000, its address may be used for B_{M_1}, B_{M_2}, or B_{M_3} whenever one of the addresses M_1, M_2, or M_3 is not to be modified but at least one of the others is. By setting $s = d_0 = 0$ in the word addressed $L + 1$, the computer will stop if this word enters the control register. However, these digits in the second word of a B-boxed command may be left to the discretion of the programmer. A similar construction of B-box commands for a two-address computer is assumed in the description for these commands in Appendix II.

Assuming the use of B-boxes, the coding for the flow diagram of Fig. 4-2-1 is given in Table 4-4-2. In coding this routine, it is again assumed that the a_j are stored in their proper successive memory locations before the routine is used, and n is now stored as $\bar{n} = 10^{-10}n$. In boxes 0.0, 0.1, and 0.2, s is set to zero before starting the sum; n is extracted into the third address of the B-box SET command; and this SET command sets the limits of the B-box so that $j' = j - 1$ runs between the values 0 and $n - 1$, inclusive. In box 1.0, the address of a_1 is incremented by the B-box so that the a_j are added in sequence to s. In box 2.0, the B-box is

TABLE 4-4-2. Three-address Code for Flow Diagram of Fig. 4-2-1 Using B-box Commands

Box no.	Order symbol: Operation	M_1	M_2	M_3	Branch	Memory location	Order code: Operation	M_1	M_2	M_3
	← 0 →					000	+0	000	000	000
0.0	ADD	0	0	S	Enter	001	01	000	000	009
0.1	XTR	$\bar{N}$	D_1	$C_{0.2}$		002	24	011	010	003
0.2	SET	000	B	$\{N\}$		003	27	000	012	{000}
1.0	ADD	S	A_1	S		004	11	009	013	009
	00	000	B	000		005	00	000	012	000
2.0	INC	001	B	$C_{1.0}$	1.0	006	28	001	012	004
4.0	PRT	001	S	M(STP)		007	42	001	009	008
	STP	000	000	000		008	00	000	000	000
	← s →					009	← s →			
	← d_1 →					010	+0	000	000	111
	← $\bar{n} = 10^{-10}n$ →					011	← $\bar{n}$ →			
	← B-box B →					012	← B-box B →			
	← a_1 →					013	← a_1 →			
	← a_2 →					014	← a_2 →			
	⋮						⋮			
	← a_n →					$n + 12$	← a_n →			

incremented by one, and a test is performed to see if the incremented value indicates that the sum is completed.

The preceding problem could have been coded using the DEC command by having n prestored as $\bar{n} = 10^{-4}n$ and by having $d_1 = +0\ 111\ 000\ 000$. By changing the two-word command of box 1.0 to

$$\begin{cases} \text{ADD} & S & A_0 & S \\ 00 & 000 & B & 000 \end{cases}$$

where A_0 is one less than the address of a_1, and by changing the SET command of box 0.2 to

$$\text{SET} \quad N \quad B \quad 000$$

Then the command

$$\text{DEC} \quad 001 \quad B \quad C_{1.0}$$

would be used in place of the INC command of box 2.0. Such a coding of the problem would sum the a_j in reverse order.

This simple example shows how a B-box may be used in modifying addresses in an order. Since there is only one address modified, the

saving of effort in programming and coding is not too apparent. However, the saving is more evident in a calculation such as

$$s = \sum_{i=1}^{n} [c_i + a_i b_i] \tag{4-4-1}$$

where one B-box may be used to increment three addresses in two commands.

By letting $s_0 = 0$ and

$$s_j = \sum_{i=1}^{j} (c_i + a_i b_i)$$

for $1 \leq j \leq n$, it follows that

$$s_j = c_j + a_j b_j + s_{j-1}$$

and for $j = n$, $s_j = s$. By assuming that the a_j, the b_j, and the c_j are stored in three sequences in the main memory and by changing box 1 of the flow diagram of Fig. 4-2-1 to that of Fig. 4-4-1, the flow diagram for this problem is obtained. In coding, the orders involving variables with subscripts are B-box tagged by adding one to the first digit of the instruction, and a second word giving the appropriate B-box addresses is added to the order. The incrementing and testing of the subscript (in the B-box) is performed by a single order. The coding of arithmetic commands to increment the address of commands involving variables with subscripts is eliminated. The net result of using B-boxes is not only a savings in coding effort, but a savings in memory space required for the routine and a reduction in the number of coding errors.

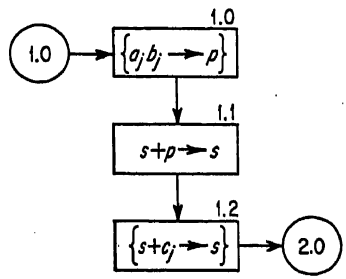


FIG. 4-4-1. Change of Fig. 4-2-1 to form $s = \sum_{i=1}^{n} (c_i + a_i b_i)$.

The problem of Eq. (4-2-1), if enlarged to that of finding a set of sums s_k where

$$s_k = \sum_{i=1}^{n} a_{ki}$$

and

$$1 \leq k \leq N$$

may be coded using two B-boxes where a command for one B-box is modified by the other B-box. Let us assume that the a_{ki} are stored in memory in sequence; that is, the a_{ki} are stored in successive cells in the

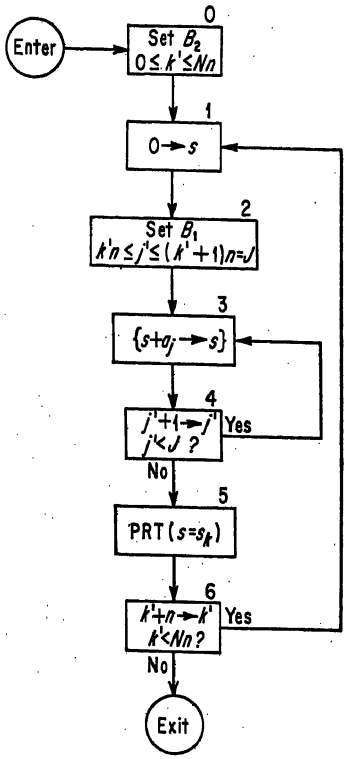


FIG. 4-4-2. Flow diagram for $s_k = \sum_{i=1}^{n} a_{ki}$.

order $a_{11}, a_{12}, \ldots, a_{1n}, a_{21}, a_{22}, \ldots, a_{2n}, \ldots, a_{N1}, a_{N2}, \ldots, a_{Nn}$. We redesignate this sequence as the sequence, with a new notation,

$$\{a_j\} \qquad 1 \le j \le Nn$$

i.e., $a_1, a_2, \ldots, a_n, a_{n+1}, a_{n+2}, \ldots, a_{2n}, \ldots, a_{(N-1)n+1}, a_{(N-1)n+2}, \ldots, a_{Nn}$. Thus

$$s_k = \sum_{j'=k'n}^{(k'+1)n-1} a_{j'+1}$$

where $k' = k - 1$ and $j' = j - 1$. The flow diagram for this routine is given in Fig. 4-4-2 where it is assumed that N and n are predetermined constants. The coding of interest is that concerned with boxes 0, 2, and 6. In box 0, B-box B_2 is set with initial value 0 and final value Nn; i.e., the code is

$$27 \quad 000 \quad B_2 \quad Nn$$

In box 2, B-box B_1 is set with initial value 0 and final value n, and this command is modified by B-box B_2; i.e.,

$$\begin{cases} 37 & 000 & B_1 & n \\ 00 & B_2 & 000 & B_2 \end{cases}$$

In box 6, B-box B_2 is incremented by the amount n; i.e.,

$$28 \quad n \quad B_2 \quad C_{1.0}$$

4-5. Indirect Addressing

The address modification scheme described in this chapter is a variation of systems that are used in some computers. A general description of address modification may be found in the article "Programming for a Machine with an Extended Address Calculational Mechanism."[1]

The address modification scheme may be used for indirect addressing; that is, for designating an address which is the location of the operands

[1] Heinz Schecher (translated by John W. Carr III), Programming for a Machine with an Extended Address Calculational Mechanism, *Commun. Assoc. Computing Machinery*, vol. 2, no. 6, pp. 32–38, June, 1959.

or connector address. This is indicated by the following example

$$\begin{cases} 32 & M_1 & M_2 & 000 \\ 00 & 000 & 000 & L \end{cases}$$

This B-modified TRA order designates a transfer if the condition is met, i.e., if $C(M_1) \geq C(M_2)$. The transfer is to the cell the address of which is the first address in location L. Two of the many uses of indirect addressing are for designating the location of a return address for exit from a subroutine to the main program or for designating other variable connector addresses obtainable by reference to a table of connector addresses.

4-6. Remarks

It is often of interest to the programmer to know the function of the digits and bits within a digit of a word. When s is represented by a binary-coded hexadecimal digit, we have already pointed out that two configurations of bits for s not recognizable as binary-coded decimal digits are used for the plus and minus sign. Thus, the computer may use the sign digit to distinguish between order words and data words and, if desired, may be constructed to set an alarm whenever a data word enters the control register. Let s be represented by

$$\sum_{i=0}^{3} s_i 2^i \qquad \text{or} \qquad s_3 s_2 s_1 s_0$$

where the s_i are bits; then s is the sign of a data word (or a word is a data word) if $s_3 = s_1 = 1$ since + (plus) is represented by 1010 and − (minus) is represented by 1011. For data words, the word is positive if $s_0 = 0$ and negative if $s_0 = 1$. Thus, s_0 may be considered as the sign toggle for data words. Noting that s_2 is not used so far in data words, we may make use of it and expand the meaning of data words. For example, if $s_2 = 0$, the digits $d_0, d_1, \ldots, d_9$ are interpreted as decimal digits; and if $s_2 = 1$, the digits are paired so that $d_{2i}d_{2i+1}$ for $i = 0, 1, 2, 3,$ and 4 represent a single character allowing the storing of alphabetical, numerical, and special characters in the computer which may be recognized by the input and output units. Thus, the binary representation $s = 1010$ may be used for + for a positive digital data word, the binary representation $s = 1011$ may be used for − for a negative digital data word, the binary representation $s = 1110$ may be used for $\oplus$ for a positive data word containing characters, and the binary representation $s = 1111$ may be used for $\ominus$ for a negative data word containing characters. For order words, either $s_3 = 1$ and $s_2 = s_1 = 0$, or $s_3 = 0$. Also, for order

words, $s_0 = 0$ means that the command is not modified by a B-box, and $s_0 = 1$ means that the command is modified by a B-box. Thus, s_0 may be considered as a B-box toggle for order words. Similarly, s_8 may be considered as the floating point toggle for arithmetic order words.

Often in the programming and coding of a routine it is desirable to have a counter which keeps track of the number of times an operation or a sequence of operations is performed. A B-box may be used for such a counter. The SET command is used to place the initial and final value (if used) in the counter; the INC or DEC command is used for counting (up or down) and for testing (if desired) to see when the count is completed. That is, in some cases one desires to know how many times the computer performs a sequence of operations, and in other cases it is desired that the computer perform a sequence of operations a fixed number of times. Such uses of a B-box need not involve the modifying of addresses in order words.

By increasing the word length of a three-address computer to a sign digit and 13 decimal digits, the main memory may be increased to 10,000 words without changing the basic order word structure and the type of B-box used. That is, for order words, s and d_0 would represent the operation code, and the remaining 12 digits would be used for the three 4-digit addresses.

In general, in the following chapters the detailed flow chart will not indicate whether or not a B-box is to be used. For example, the flow diagram of Fig. 4-2-1 has been considered as an adequate description of the problem when a B-box is used. However, when programming and coding a complicated routine for a computer with B-boxes, it may be desirable to replace parts of the detailed flow diagram with computer flow diagrams indicating the B-box operations to be coded, as in Fig. 4-4-2.

PROBLEMS

4-1. Recode Prob. 3-4 using B-boxes (*a*) for the three-address computer and (*b*) for the two-address computer.

4-2. Recode Prob. 3-5 using B-boxes (*a*) for the three-address computer and (*b*) for the two-address computer.

4-3. Code Prob. 1-4 using B-boxes (*a*) for the three-address computer and (*b*) for the two-address computer.

4-4. Code Prob. 1-6 using B-boxes (*a*) for the three-address computer and (*b*) for the two-address computer.

4-5. Develop a dynamic memory chart for a program for Fig. 4-4-2. Show the contents of the memory cells assigned as the index registers B_1 and B_2 after each pass through the flow chart boxes 0, 2, 4, and 6. Also show the contents of the control register when it contains the modified command for box 2.

5

SUBROUTINES

5-1. Introduction

Many computer routines have considerable complexity. For such routines it is advantageous to separate the parts which are essentially independent of one another. The remainder of the routine then contains the part which is concerned with the organization of the independent parts into a unified routine. The independent parts of the routine are called *subroutines*, and the organizational part of the routine is called the *main routine*.

The principal advantage for this approach is that in designing the main routine the inner structure of the subroutine may be considered irrelevant. The subroutine is to be viewed from its functional characteristics alone. Further, in the detection of coding errors the subroutines can be tested independently.

The independent nature of subroutines contributes to their general applicability. The existence of subroutines for standard processes can be an important factor in the economics of automatic computation. For many computations, savings of over half of the coding effort required can be effected by the incorporation of previously written and tested subroutines. This has led computer users to develop collections of subroutines which have been written for general applicability and are thoroughly tested to assure fault-free operation. These collections are called *subroutine libraries*. One of the first groups to develop a library of subroutines was the staff of the Mathematical Laboratory, University of Cambridge. The book entitled "The Preparation of Programs for an Electronic Digital Computer" by Wilkes, Wheeler, and Gill[1] is mainly a

[1] M. V. Wilkes, D. S. Wheeler, and S. Gill, "The Preparation of Programs for an Electronic Digital Computer," 2d ed., Addison and Wesley, Inc., Cambridge, Mass., 1957.

description of the subroutine library developed for their computer, the EDSAC. Their book is the first book solely on programming for digital computers and also contains much useful information on the design and testing of routines.

In this chapter the techniques of design and use of the subroutines are considered. Emphasis is placed on procedures for incorporating a subroutine into a code, i.e., the procedures for linking the subroutines to the rest of the program. The term *calling sequence* has been coined as a generic term for the sequence of orders, addresses, and parameters contained in the main routine which are required to form the linkage with the subroutine. The term *executive routine* is sometimes used as a synonym for *main routine*. The use of the term *subroutine* has become diluted and is sometimes used to refer to any well-defined part of a program.

5-2. Subroutines for the Evaluation of a Function of One Variable

Subroutines for the determination of a number, the value of which is determined by another number, occur frequently in computer applications. Let $y = f(x)$ denote this functional relation. In addition to the calculation of the function, programming considerations for such subroutines include: determination of a procedure for supplying the subroutine with the input variable x (alternately the memory location of x or the memory location of the location of x could be supplied); determination of a procedure for incorporation of the result y in subsequent calculation; and determination of procedures for transfer of control to the subroutine and transfer of control from the subroutine to the subsequent program. These considerations constitute the procedures for incorporating a subroutine in a routine. Subroutines designed for general application should be as convenient to use as is consistent with general computation requirements. That is, a subroutine which is simple to incorporate in a program may take more computer time than one which requires a longer linking code in the main routine. In such event, two or more versions of the subroutine may be put in the library.

5-3. Open Subroutines

A subroutine which is to be fitted into a routine so that no transfer of control is necessary is called an open subroutine. A flow diagram for a single-address computer routine using an open subroutine to calculate $y = f(x)$ is illustrated in Fig. 5-3-1.

As is shown in this flow diagram, the upper accumulator A_U (or lower accumulator A_L) of a single-address computer can be used for temporary storage of the input variable and subsequently the result. This use of standard locations for the subroutine variables facilitates their design and use. Since open subroutines are incorporated directly into the routine (without transfer commands to and from the rest of the program), their use in more than one part of the routine generally requires duplication of the subroutine.

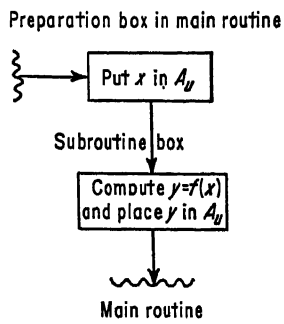


FIG. 5-3-1. Flow diagram of an open subroutine for the evaluation of a function of one variable.

To minimize the chance for error, subroutines frequently include safety features. These may involve saving and restoring the contents of arithmetic and index registers. Subroutines may also check for consistency in input data and may have computation checks. In such cases the subroutine may initiate printing of an *error message* indicating the illegal subroutine use or computational error involved.

Some computer groups use the term *subroutine library* to refer to subroutines stored in a magnetic tape or other form of auxiliary memory. *Program library* is then used to describe the collection of less frequently used subroutines.

5-4. Location of Subroutines

In most subroutines some of the commands will refer to numbers or commands contained in the subroutine. Thus the addresses in these commands are related to the location of the subroutine in the memory. For general application it is important that the subroutines be written in a form suitable for relocation in an arbitrary part of the memory. The modification of the subroutine for such relocation is usually accomplished by a computer routine. Routines which perform the relocation required for subroutines are discussed in Chap. 8 entitled Some Aspects of Automatic Programming.

5-5. Subroutine Parameters

Parameter values used by the subroutine which are fixed during program assembly and not changed during the computations are called *preset parameters*. There may be a requirement to occasionally change a parameter value during the computation and for the computation

routine to supply the subroutine with the values of these particular parameters. Such parameters supplied by the main routine are called *program parameters* for the subroutine.

5-6. Closed Subroutines

Subroutines which are incorporated in a routine by the use of transfer commands are called closed subroutines. A flow diagram for a three-address computer of Appendix I, illustrating the incorporation of a closed

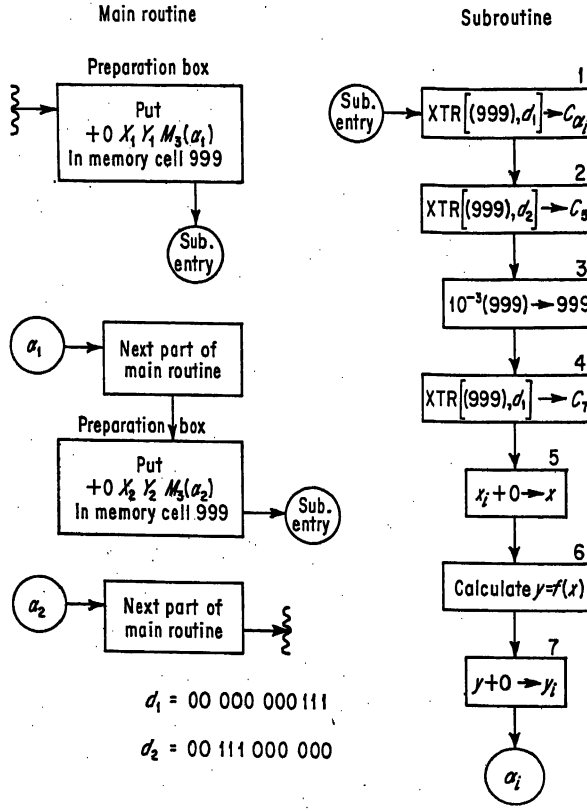


FIG. 5-6-1. Illustration showing the incorporation of a closed subroutine in a main routine.

subroutine for evaluating $y = f(x)$, is shown in Fig. 5-6-1. This figure illustrates the use of one subroutine for computation of $y = f(x)$ to compute $y_1 = f(x_1)$ and $y_2 = f(x_2)$ in two different parts of the routine.

In this flow diagram the addresses of x_1, x_2, y_1, and y_2 are X_1, X_2, Y_1, and Y_2, respectively for the two uses. The addresses for the next command in the main routine are designated by $M_3(\alpha_i)$, $i = 1, 2$, where α_i is the variable connector. *Main routine* refers to the part of the computation routine which does not include the subroutine (or subroutines). In this flow diagram, the command for the variable connector α_i is designated by C_{α_i}, and the command for the jth box in the subroutine is designated by C_j. The extract command XTR $M_1M_2M_3$ is shown in the flow diagram as $\text{XTR}(m_1,m_2) \rightarrow m_3$. Also, (999) represents the contents of the memory cell addressed 999. Preceding the entry to the subroutine, a word with the three addresses X_i, Y_i, and $M_3(\alpha_i)$ required by the subroutine is placed in a standard location (in this case cell 999). The first digit of this word is not used and is represented in the preparation box by a zero. The subroutine separates the required addresses and places them in the appropriate commands within the subroutine; that is, in box 1 the return address to the main routine is extracted from the contents of cell 999 using the extractor word designated as d_1 and placed in the M_3 address of the UCD command for the variable connector α_i at the end of box 7. In box 2 the address for x_i is extracted from the contents of cell 999 and placed in the M_1 address of the command for box 5. In boxes 3 and 4 the contents of cell 999 are shifted three digits to the right and re-stored there, and then the shifted address for y_i is extracted into the M_3 address of the ADD command for box 7. One of the principal advantages of this system is that it allows the incorporation of (a single copy of) the subroutine at several places in the main routine.

If a subroutine is to be used at only one place in a routine, then an open subroutine is more economical for this application than a closed subroutine as computer time and memory are required for the transfer commands linking the closed subroutine to the main routine. In other applications the closed subroutine may be more economical because of a saving in memory requirement.

5-7. Automatic Return to the Main Routine

Because of the large number of components in an automatically sequenced computer and the prodigious technical difficulties encountered in the construction and test of the first computers, emphasis was first placed on the development of computers with a minimum number of automatic features. Some of these early computers did not even perform multiplication and division automatically. These operations were accomplished through the use of closed subroutines. In many applica-

tions control is returned from a closed subroutine to the command following the command which transfers control to the subroutine. That is, if the transfer command from the main routine is in cell n, control is returned from the subroutine to cell $n + 1$. In several of the computers designed after 1950, provision was made to save the address of the previous command in a special register. The first command of the subroutine may then save this address for the location of the return to the main program. This makes the process for returning to the main routine automatic. B registers can also be used for the same effect, one of the computer's B registers being set to the address for return to the main routine before the transfer to the subroutine is made.

5-8. Interpretive Subroutines

An early discovery in the use of automatic digital computers, particularly in scientific applications, was that the cost of programming a computation might exceed the cost for performing the calculations on the computer. For this reason early consideration was given to means of reducing programming costs. One method for effecting a reduction in programming effort at the expense of an increase in computer time uses a computer subroutine designed to simulate the operation of a computer with operational features not contained in the basic computer. To this end a subroutine is used which decodes (interprets) computer words (or perhaps sets of more than one word) as commands and numbers in an expanded order code. For example, a single-address computer with fixed point arithmetic could be used to simulate a three-address computer with both fixed and floating point arithmetic.

An *interpretive subroutine* is a subroutine whose operation is controlled by a sequence of computer words (or perhaps sequences of sets of more than one word).

A flow chart for an interpretive subroutine is shown in Fig. 5-8-1. This flow chart illustrates the basic features of most interpretive subroutines. The boxes of the subroutine are numbered serially from 1 to $L + 3$. The box in the main routine which precedes box 2 of the subroutine specifies the location N of the initial word in the sequence of computer words to be decoded. The memory cell M acts as a control counter for the interpretive subroutine. Thus, when control is transferred to the subroutine from the main routine, the first word interpreted is fetched from cell N. In box 3 the word to be decoded is called a *pseudo command* to differentiate it from the computer commands. The exit from box 3 is a variable exit. This variable exit transfers control to

one of the L operations (boxes 4, 5, 6, . . . , $L + 2$, $L + 3$) performed by the interpretive subroutine. If the addresses for the first command of each of the L operations is a simple function of the pseudo-command code, then the address for this transfer can be computed. In other cases it is generally advisable to store the addresses in a table in the sequence

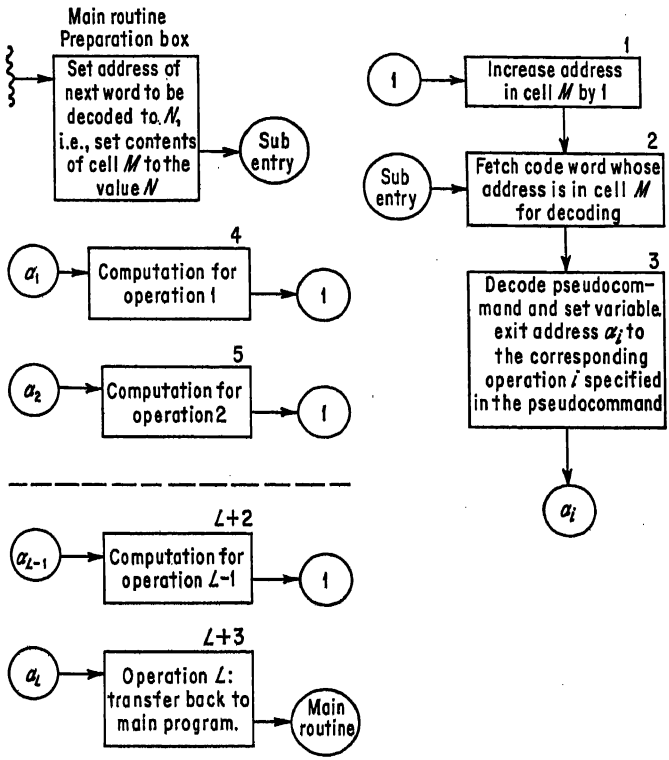


FIG. 5-8-1. Flow chart for an interpretive subroutine.

corresponding to that of the pseudo-command codes or to use a table of transfer commands.

The operation shown in box $L + 3$ is included to allow a change back to coding in the basic computer operations. After one of the other operations, i.e., box 4, 5, 6, . . . , or $L + 2$, the next step is the preparation of the address for the next word to be decoded. This is shown as box 1 in the flow chart. For some interpretive subroutines it is advisable to include provision to change the decoding sequence. This can be accom-

plished by having one of the operations change the value of the address contained in cell M. It is also frequently advantageous to include provision for other pseudo commands of a logical nature. For example, iterative computational procedures require logical operations. If logical pseudo commands are included in the interpretive routine then it is not necessary to transfer back to normal command coding to accomplish these operations. Several interpretive subroutines have been written with the intent that all of the coding for a problem be done with pseudo commands.

5-9. Subroutines for More than One Process

It sometimes happens that parts of the codes for two or more calculations are the same. A savings in memory requirements may result when the codes for these processes are combined. For example, one subroutine may be used to calculate either $\cos x$ or $\sin x$. In this case the mathematical identity $\cos y = \sin(\pi/2 - y)$ may be used to establish the common part of the code. The flow chart which follows illustrates how a multiple entry system could be used for such a sine-cosine subroutine. The use of multiple entries can, of course, be extended to more complicated overlapping processes.

Other schemes for combining some computations in a common routine are also feasible. Variable connectors can be used to prepare the routine for one of the computations it is designed to execute. The section on interpretive routines illustrated one such use of variable connectors. In another approach, parameters are used to particularize the calculations of a multipurpose subroutine. Here the function evaluated by the subroutine is determined by specific values of the parameters instead of by different transfer paths to or within the subroutine or both. In the sine-cosine calculations a routine which evaluates $\sin(x + b)$ could be used. Specific values for the parameters would then determine whether the sine, cosine, or a linear combination of the two is evaluated. Atta and Sangren[1] have used this parameter approach in the formulation of a subroutine which can be used to evaluate Bessel functions, gamma functions, Laguerre

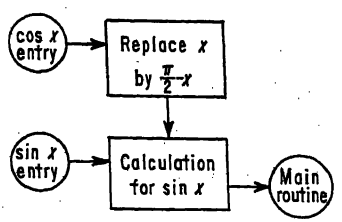


FIG. 5-9-1. Example of a multiple-entry subroutine.

[1] E. E. Atta, and W. C. Sangren, Calculation of Generalized Hypergeometric Series, *J. Assoc. Computing Machinery*, vol. 1, pp. 170–172, 1954.

polynomials, and some other special functions of generalized hypergeometric type.

5-10. Levels of Subroutines

In some applications it is advantageous to have subroutines which refer to other subroutines. Consider, for example, a calculation for which the arccosine function is required. The approximation

$$\theta = \cos^{-1} x \doteq (a_0 + a_1 x + a_2 x^2 + \cdots + a_7 x^7)\sqrt{1 - x} \qquad 0 \leq x \leq 1$$

developed by C. Hastings[1] provides an elementary procedure for determining θ with an error less than 3×10^{-8}. Many routines requiring the arccosine function also require the square-root function. In such cases there will be a savings in memory requirement if the arccosine routine utilizes the square-root routine as a separate subroutine. Then the square-root subroutine may also be used as required in the rest of the computations.

A subroutine which does not refer to another subroutine is called a subroutine of level 1. Similarly, a subroutine is said to be of level $n + 1$ if it refers directly to one or more subroutines of level n and perhaps subroutines of level less than n. Applications using subroutines of level 5 are not uncommon.[2]

The two basic considerations for incorporation of a multilevel subroutine into a program are those of linkage commands and temporary storage requirements. The commands linking the subroutines together require either direct or indirect (such as location of a location) specification of the location of the subroutines. Assembly of a routine written as several parts requires the generation of the appropriate linking commands. Assembly routines to facilitate this linking are described in Chap. 8. These assembly routines use relative addressing and free addressing systems to automatically construct the linkage. The subroutines are written with free or relative addresses which may refer to other subroutines. The assembly routine then uses a cross-reference table to determine the actual address used in the linkage commands.

Since a subroutine of level 1 does not refer to any other subroutine, there is no restriction on the use of the temporary storage required for the subroutine by other subroutines. A subroutine of level 2 may require temporary storage that may not be used by the subroutine to which it

[1] C. Hastings, "Approximations for Digital Computers," Princeton University Press, Princeton, N.J., 1955.

[2] See Wilkes *et al.*, *op. cit.*

refers. Thus in the description for a multilevel subroutine it may be desirable that the temporary storage requirement for the subroutine be categorized into levels of availability. Reservation of the maximum temporary storage specified for each category of the subroutines used in a computation would provide enough temporary storage. This simple rule in some cases will specify more storage than is necessary. Instead, the minimal temporary storage requirement for a routine should be determined from an examination of all the possible interrelations of temporary storage requirements for the interdependent parts of a routine.

5-11. Subroutine Library

The subroutines for a computer are usually organized into a collection called the *subroutine library*. The subroutine library may consist of the following: cross-referenced listings of the subroutine titles and identifying

TABLE 5-11-1. Possible Subroutine Categories for an Engineering Computation Center†

1. Utility routines
   - *a.* Assembly
   - *b.* Compiler
   - *c.* Program error detection
2. Data processing
   - *a.* Collation, sorting
   - *b.* Output
   - *c.* Tape handling
3. Interpretive routines
4. Mathematical functions: elementary and special
5. Matrix programs
6. Differential and integral equations
7. Fourier analysis
8. Statistics, Monte Carlo
9. Other mathematical programs
10. Engineering applications
11. Maintenance routines

† A more complete list is given by W. L. Frank, Mathematical Subroutines for the Univac Scientific Computer—A Survey, *Computation and Automation*, vol. 6, no. 9, September, 1957.

number, abstracts for the subroutines, detailed descriptions of the subroutines, master copies of the subroutine code on tapes or cards for computer input, and duplicates of these tapes or cards for actual use of the computer.

A programmer needing a subroutine first examines the subroutine listings to find the subroutines that seem to meet the requirements for his

computation. Examination of the abstracts provides additional information such as storage requirements, the amount of computer time used by the routine, accuracy, etc. These abstracts are usually adequate for the programmer to determine whether the subroutine is appropriate.

Master copies of the subroutine tapes, cards or other input media are only used in a duplicator to prepare copies for use on the computer. Since it is usually difficult or time-consuming to obtain an error-free copy of the routine by manual methods, some computation laboratories keep the master copies in a locked file to ensure against loss or destruction of all the error-free copies.

A list of types of subroutines that might be used in an engineering computation center is shown in Table 5-11-1.

The value of a subroutine library is determined by the standards set for the preparation, test, and description of the subroutines. The repeated use of the subroutines makes it economical for the programmer to consider several alternate computation methods or coding sequences to determine efficient routines. For an elementary function such as e^x, the computation methods examined might include series expansions, recursion relations, Chebychev polynomial approximation, continued fraction expansion,[1] and table look-up, the method used for the subroutine being chosen from practical consideration of accuracy, computation time, and memory requirement. For each of the computation methods the accuracy, speed, and space will depend to some extent on the coding sequence. An error in a subroutine contained in the subroutine library can cause a large waste of computer time as well as programmer consternation. In general, the most economical testing of a routine is done by the writer testing the routine soon after the program has been written. Thus, only thoroughly checked subroutines should be put in the library. The library should contain a detailed description for each of the subroutines with sufficient information for programmers to understand the operation and analysis for the subroutine, and when required, to modify the subroutine. Such detailed descriptions may contain the title, purpose, range, accuracy, restrictions, storage requirements, speed, entry, exit, and other procedures for using the subroutine, equipment specifications (e.g., subroutine uses two tape units), mathematical analysis, flow chart, code and a description of how the subroutine was tested. Programmers should be continually encouraged to add to the subroutine library whenever they have written a program that has general applications.

[1] N. Macon, A Continued Fraction for e^x, "*Math. Tables Aids Comput.*" vol. 9, no. 52, pp. 194–195, October, 1955.

Subroutine libraries vary with the computer used. Most computation groups develop extensive libraries of subroutines and complete programs. Many of these programs are very useful and contain complex and difficult-to-program computation algorithms. Thus a change in the computer may involve extensive and costly modification of the library of programs. Difficulties also occur in exchanging programs between different user groups. Such changes and difficulties are minimized when computer-independent programming methods are used.[1] The Association for Computing Machinery has adopted ALGOL[2] (International Algorithmic Oriented Language) as a machine-independent programming language for scientific applications. In March, 1960, the ACM started publication of *ALGOL Procedures*. These procedures are general-purpose subroutines. Although some of the initially published routines contained errors, recent issues of the *Communications* contain tested procedures which are very useful and have excellent computation algorithms.

5-12. Subroutine Descriptions

This section contains a description of a subroutine for illustration of considerations which are of interest to the subroutine user. The description starts with the frequently needed details listed in a standard subroutine description heading. The title of the subroutine is as descriptive as is practical and may contain brief statements about the restrictions on the computation. The identity number is used for indexing the descriptions and may be used by an assembly routine which automatically retrieves subroutines from a magnetic-tape file (auxiliary memory). When an identity number or symbol is used in this manner it is referred to as the *calling number*. In the identity number of the following subroutine description the 5 refers to matrix programs, .01 to vectors, and .02 to the subnumber for this routine (*cf.* Table 5-11-1). The description contains a flow diagram, Fig. 5-12-1, and code, Table 5-12-1, for use in modifying the subroutine, detection of transcription errors or locating computer malfunction. In computing the running times for the subroutine, all commands except FMR are assumed to take 0.16 milliseconds (msec). The FMR command is assumed to take $0.08(1 + 2S_m)$ milliseconds where S_m is the sum of the digits in the multiplier. By letting

[1] *Cf.* Saul Gorn, Standardized Programming Methods and Universal Coding, *J. Assoc. Computing Machinery*, vol. 4, no. 3, July, 1957.

[2] Peter Naur et al., Report on the Algorithmic Language ALGOL 60 *Commun. Assoc. Computing Machinery*, vol. 3, no. 5, May, 1960.

TITLE

Scalar Product of Two Vectors (Floating Point Arithmethic) $s = \sum_{i=1}^{n} x_i y_i$, $1 \leq n \leq 957$

IDENT. NO. 5.01.02

COMPUTER Two-address (Appendix II)

NO. OF CELLS	TEMP. STOR.	AVERAGE TIME (Entry 1)	AVERAGE TIME (Entry 3)	ACCURACY
40	3	$0.96 + 6.80n$ msec	$3.68 + 6.80n$ msec	$\lvert\epsilon\rvert \leq 5n \cdot 10^{-9} \cdot \max\lvert x_i y_i\rvert$

ENTRY

1. To compute scalar product for preset values of $M(x_1)$, $M(y_1)$, and n, place +00 0000 EXIT in A_U and transfer to "ENTRY 1," i.e., first word of subroutine.
2. To change $M(x_1) = XXXX$, $M(y_1) = YYYY$ and not change n, put +00 0000 EXIT in A_U and +00 $XXXX$ $YYYY$ in Q and transfer to "ENTRY 2," i.e., 16th word of subroutine.
3. To change $M(x_1)$, $M(x_2)$, and $n = NNNN$, put +00 $NNNN$ EXIT in A_U and +00 $XXXX$ $YYYY$ in Q and transfer to "ENTRY 3," i.e., 28th word of subroutine.

EXIT

Result left in A_U and control returned to command in main routine designated by EXIT.

Restrictions:

The subroutine is coded so n must be greater than zero. (Modification of boxes 2, 3, and 4 would allow the case $n = 0$.)

Input Format:

Subroutine is available on cards or tape for assembly by any of the assembly routines.

Equipment:

No auxiliary equipment is used. Subroutine uses one B-box.

Preset Parameters:

The values for $M(x_1)$, $M(y_1)$, and n may be assigned as preset parameters either in the assembly of the routine or in the preceding computations; *cf.* description for additional details.

Program Parameters:

The values of EXIT, $M(x_1)$, $M(y_1)$, and/or n may be changed as described in the heading above; *cf.* description for additional details.

Description:

This closed subroutine can be used in three ways to compute $\Sigma_{i=1}^{n} x_i y_i$ from the values $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$ stored in two sets of successive memory cells, the addresses of which are $M(x_1)$, $M(x_1) + 1, \ldots$ and $M(y_1)$, $M(y_1) + 1, \ldots$, respectively.

Floating point arithmetic is used and the values of x_i and y_i must be in floating point notation.

The addresses $M(x_1)$, $M(y_1)$ and EXIT and the integer n are used as program parameters in the subroutine. If the subroutine is entered at ENTRY 1, the last four digits of A_U are placed in the EXIT connector of the subroutine. If the subroutine is entered at ENTRY 2, the last four digits of A_U are placed in the EXIT connector of the subroutine; and the next to last group of four digits, $XXXX$, of Q are substituted for $M(x_1)$, and the last group of four digits, $YYYY$, of Q are substituted for $M(y_1)$. If the subroutine is entered

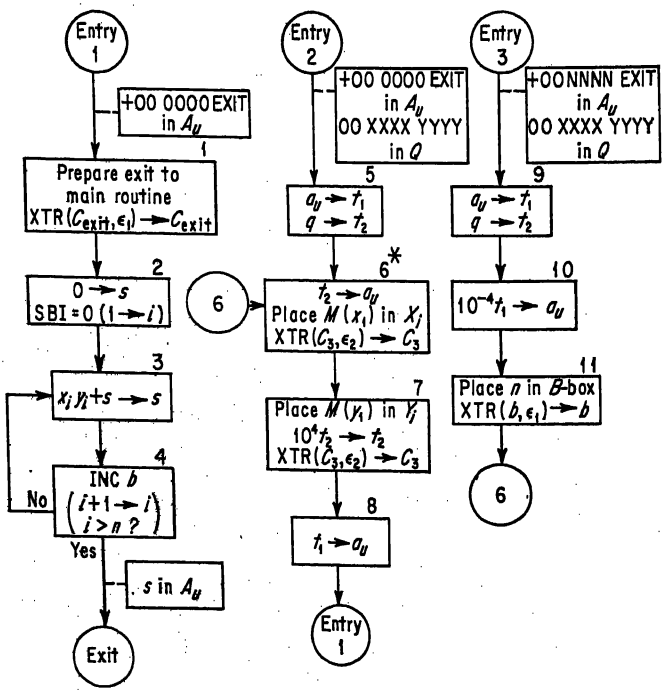
at ENTRY 3, the next to last group of four digits, $NNNN$, of A_U are substituted for the final value n in B-box B, and the last group of four digits are placed in the EXIT connector; and the next to last group of four digits in Q are substituted for $M(x_1)$, and the last group of four digits in Q are substituted for $M(y_1)$.

Time:

This subroutine operates most quickly when $M(x_1)$, $M(y_1)$, and n are not changed; i.e., for entry at ENTRY 1 of subroutine. The change of $M(x_1)$, $M(y_1)$, and n takes 2.72 milliseconds (msec). The computation time increases linearly with n and the sum Σ of the precision digits in the multipliers $y_1, y_2, \ldots, y_n$.

Accuracy:

Errors are produced in the evaluation by the floating point multiplications and additions. The rounding error in each product does not exceed $5 \cdot 10^{-9}|x_i y_i|$ unless floating point exponent overflow occurs. The error in the rounded sum will not exceed $5n \cdot 10^{-9}$ max $|x_i y_i|$.



ϵ_1 = -11 1111 0000, ϵ_2 = -11 0000 1111, b = $C(B$-box); t_1, t_2 and s are temporary storage.

Flow diagram for $s = \sum_{i=1}^{n} x_i y_i$.

* The first step of box 6 would not be here if the input parameters in A_U and Q were reversed. The input parameters have the same location for each of the three entries. This is done to reduce the possibility for error.

FIG. 5-12-1. Subroutine description (flow diagram).

(Page 2 of 3 pages)

TABLE 5-12-1. Coding Sheets for $s = \sum_{i=1}^{n} x_i y_i$ (Actual Digital Code Omitted from This Sample Description)

Box	Order symbol			Branch	Remarks
	I	M_1	M_2		
1.0	CAQ	E_1		ENTRY 1	Extractor e_1 to Q
1.1	XTR	$C_{4.3}$			Extract EXIT into UCD command
1.2	STU	$C_{4.3}$			Store new EXIT order
2.1	CAU	0			$0 \rightarrow a_U$
2.2	STU	S			Set initial sum $s = 0$
2.3	SBI	B	0000		Set initial value of B-box B to 0
3.1F	−CAQ	X_1			x_1 (as modified by B-box) $\rightarrow q$
3.1L	00	B	0000		
3.2F	−FMR	Y_1			qy_1 (as modified by B-box) $= x_i y_i \rightarrow a_U$
3.2L	00	B	0000		
3.3	FHA	S			$a_U + s \rightarrow a_U = x_i y_i + s$
3.4	STU	S			$a_U \rightarrow s$. Partial sum also in A_U
4.1	INC	B	0001	If no to 4.3	$i + 1 \rightarrow i$ and is $i < n$?
4.2	UCD	0000	$C_{3.1}$	3.1	Go to box 3
4.3	UCD	0000	[ ]	EXIT	Exit to main routine
5.1	STU	T_1		ENTRY 2	Store t_1 = +00 0000 EXIT
5.2	STQ	T_2			Store t_2 = +00 $XXXX$ $YYYY$
6.1	CAU	T_2			Place $M(x_1) = XXXX$ in $C_{3.1F}$
6.2	CAQ	E_2			
6.3	XTR	$C_{3.1F}$			
6.4	STU	$C_{3.1F}$			
7.1	CAU	T_2			$t_2 \rightarrow a_U$
7.2	SHL	0004			Shift left four digits
7.3	CAQ	E_2			Place $M(y_1) = YYYY$ in $C_{3.2F}$
7.4	XTR	$C_{3.2F}$			
7.5	STU	$C_{3.2F}$			
8.1	CAU	T_1	$C_{1.1}$	1.1	
9.1	STU	T_1		ENTRY 3	Store t_1 = +00 $NNNN$ EXIT
9.2	STQ	T_2			Store t_2 = +00 $XXXX$ $YYYY$
10.1	CAU	T_1			$t_1 \rightarrow a_U$
10.2	SHR	0004			Shift right four digits
11.1	CAQ	E_1			Place $NNNN$ in B-box B
11.2	XTR	B			
11.3	STU	B	$C_{6.1}$	6.1	
S	←	s	→		Temporary storage for partial sum
B	←	b	→		B-box B
T_1	←	t_1	→		
T_2	←	t_2	→		
E_1	−11	1111	0000		e_1
E_2	−11	0000	1111		e_2

$S_m^{(i)}$ be the sum of the digits in y_i and $\Sigma = \sum_i S_m^{(i)}$, then the time required for the n multiplications by the y_i's is

$$\sum_{i=1}^{n} 0.08(1 + 2S_m^{(i)}) = 0.08(n + 2\Sigma)$$

If the digits of the numbers are assumed to be uniformly distributed (with eight precision digits in the floating point representation), this time estimate for the multiplication would be $0.08(1 + 2 \times 4.5 \times 8)n = 5.84n$ milliseconds.

PROBLEMS

5-1. Using the power-series expansions

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!}$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

and

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

construct a flow diagram for a subroutine which can be used for evaluating e^x, e^{-x}, $\sin x$, or $\cos x$ where $0 \leq x \leq \pi/4$. Use the convergence criterion that the absolute value of the nth term be less than δ.

5-2. Using the identities

$$\sin\left(\frac{\pi}{2} - y\right) = \cos y$$

$$\cos\left(\frac{\pi}{2} - y\right) = \sin y$$

$$e^{\pi/4+y} = e^{\pi/4}e^{y}$$

and

$$e^{-(\pi/4+y)} = e^{-\pi/4}e^{-y}$$

construct a flow diagram for a second-level subroutine which can be used for evaluating e^x, e^{-x}, $\sin x$, or $\cos x$ for the enlarged interval $0 \leq x \leq \pi/2$. Consider $e^{\pi/4}$, $e^{-\pi/4}$, $\sin \pi/4 = \cos \pi/4 = 1/\sqrt{2}$ to be known constants. Incorporate the subroutine of Prob. 5-1 as part of this flow diagram.

6

PARALLEL AND SERIAL MODES OF COMPUTER OPERATION AND OPTIMUM CODING

6-1. Introduction

Digital computers are classified by their modes of operation as well as by the number base in which the arithmetic operations are performed and in which the words are represented. The basic modes of operation are *serial* and *parallel*. These terms describe the manner in which the computer words are transmitted from one unit of the computer to another; e.g., from the memory to the arithmetic unit, and the manner in which the arithmetic unit operates on its operands. If there is a single transmission line over which a computer word travels, then the digits of the word are transmitted sequentially, the least significant digit first and the most significant digit last. Generally, when the digits of a word are transmitted serially, the arithmetic unit performs its operations serially digit by digit. For example, in the addition operation the sum digit and the carry bit from the addition of the least significant digit of the augend and the least significant digit of the addend are first formed; then the sum digit and the carry bit from the addition of the next least significant digit of the augend, the next least significant digit of the addend, and the carry bit from the least significant digit position are formed, etc. Similarly, the digits of a word are transmitted from the memory with the least significant digit first. Such a computer is referred to as a completely serial computer.

If all of the bits of a word are transmitted simultaneously from one unit of a computer to another (i.e., a distinct transmission line between units of the computer exists for each bit), and if the arithmetic unit per-

forms, insofar as is practicable, its operations on all bits simultaneously, the computer is said to have a parallel mode of operation.

Figure 9-6-17 in Chap. 9, Mathematical Aid for Programming and Computer Design, is an example of a logical design of part of a parallel adder. However, either for mathematical or electronic reasons, the successive carry bits are not added simultaneously as they are formed, but are added either sequentially or by successive parallel additions to the sum digits of the addend.[1]

Some computers combine both modes of operation. For example, a binary-coded decimal computer may operate on the decimal digits in the serial mode; and the bits, forming the code for the decimal digits, may be transmitted or operated upon in a parallel mode. In designing a computer, the choice between parallel or serial mode of operation is made as a compromise between obtaining a high-speed computer and obtaining a less expensive computer. A parallel computer may be made to operate at higher speeds than a serial computer since, in general, it performs its operations on all digits of a word simultaneously, whereas a serial computer operates on each digit sequentially. The ratio of speed of a parallel computer to a serial computer may be almost the ratio of the number of bits per word to one. On the other hand, the number of component parts and transmission lines between units of a parallel computer is greatly increased over that number required for a serial computer. However, it should be noted that the registers and the memory of a serial computer must also store all digits.

6-2. Memories of Parallel Computers

In order that each digit of a word may be withdrawn and entered into the memory of a parallel computer simultaneously, memories are constructed with equal accessibility for all digits of all words. These memories are referred to as *random access* memories. That is, a word may be withdrawn from or entered into any memory cell with equal access time so that the memory look-up time for a random sequence of a given number of words is the same as for the look-up time in any other order of look-up. At the time of this writing, most high-speed, random-access memories are being constructed of *magnetic cores*. A magnetic core is a small ring-shaped electromagnet made of ferromagnetic material. The

[1] There is a procedure for simultaneous additions of intermediate carries that permits somewhat faster operation than the procedure in which carries are performed sequentially. This procedure is described by B. V. Bowden, "Faster than Thought," Sir Isaac Pitman & Sons, Ltd., London, 1953.

direction of magnetic polarization of the magnet is used to indicate the bit stored. In such a memory a magnetic core is used for each bit of each word whether the word consists of coded decimal digits or is a straight binary word. The cores are physically arranged in a three-dimensional array with two of the dimensions used to determine the address of the word and the third dimension provides the bit location within the word.

Another random access memory presently in use is called the *Williams*[1] *tube* or *electrostatic* memory. Such a memory is constructed either of standard cathode-ray tubes, which have been properly adapted for the purpose, or of special cathode-ray tubes designed for memories. Bits are stored as electric charges at designated locations on the inside surface of the front of the cathode-ray tube. The bit location may be charged with two different patterns allowing the representation of one or zero. The sensing and writing of a bit in a location is accomplished by positioning the electron beam to that location and performing a series of electronic operations which will accomplish the desired operation. If n bits may be stored on the face of a particular Williams tube, then by having m tubes, the basic module of memory is n words of m bits each. For parallel operation, one tube contains the first bits of all words, another tube contains the second bits of all words, etc., and the transmitting of a word from or to memory is accomplished by positioning the beam of each tube to the same bit position. In this manner the digits of a word may be entered or withdrawn in a parallel mode of operation.

Random access or parallel memories, when properly constructed, present no new problems in programming or coding beyond those already discussed in the general techniques of Chap. 3.

6-3. Memories of Serial Computers

There are two types of main memory, the *magnetic drum*[2] and the *delay line*, presently used with serial computers. The principal parts of a magnetic-drum memory are a magnetic drum and reading and writing heads. A magnetic drum is a cylindrical drum, the surface of which is composed of magnetic material. The reading and writing heads are small electromagnets shaped so that there is a small gap between the poles of the magnet. Often a single electromagnet is used for both reading and writing and is referred to as a read-write head. For our pur-

[1] S. B. Williams, "Digital Computing Systems," McGraw-Hill Book Company, Inc., New York, 1959.

[2] Magnetic-disk memories are not discussed here since their operation is the same in principle as that of magnetic-drum memories.

poses, we can assume that the read-write heads are mounted along a straight line parallel to the axis of rotation of the drum with the gap near the surface of the drum as shown in Fig. 6-3-1 (*cf.* Fig. 7-1-1). By having the drum rotate, the electrical pulse representing a bit can be transformed by the read-write head into a small magnetized area in the magnetic surface. Conversely, a small magnetized area passing directly under the gap of the electromagnet will cause the read-write head to emit an electrical signal. That part of the surface of the drum which passes under a read-write head is referred to as a *track*. Words, digits of the words, and often bits of coded digits are stored serially in the same track of the drum.

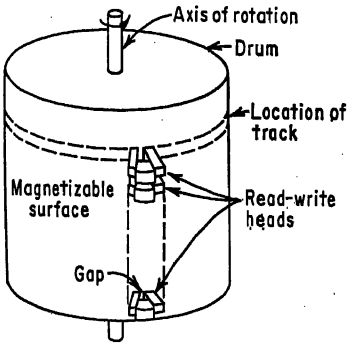


FIG. 6-3-1. Basic parts of a magnetic drum memory.

Lengths of most drums are between about 2 and 24 inches, and their diameters are between 2 and 14 inches. The rotation speed of drums is between 1,000 and 100,000 rpm. By storing 200 words in each track, a drum with 20 tracks will have a memory capacity of 4,000 words. If the words are ten binary-coded decimal digits, using four bits for each digit, and a sign digit, then each word has at least 41 bits, and a track would contain 8,200 bits. Such a drum rotating at 3,600 rpm or 60 rps would cause a magnetized area representing a bit to pass under a read-write head every 2.03 microseconds (2.03×10^{-6} seconds). Since there may be space between each bit, the pulse time for a single bit would be nearer 1 microsecond. If the drum diameter is 10 inches, then the distance between centers of the magnetized areas for bits is 0.003834 inch, and the gap between poles of the electromagnet of a read-write head would be considerably smaller, say 0.0003 inch. The size of the gap and the length of the magnetized areas for such a drum can be increased without changing the memory diameter by recording the bits of each decimal digit in parallel; e.g., by recording the bits in a set of four tracks. Then, each subtrack will contain $200 \times 11 = 2{,}200$ bits and will cause a bit to pass under a read-write head every 7.58 μsec (microseconds). In addition to the memory tracks, there are one or more timing tracks on the drum. One such track may be reserved for a pulse to locate the first word (i.e., the word with the smallest address) in each track. Another is used to designate the word positions; a third to designate the digit locations within the words; and a fourth, if the drum is completely serial, to

show the location of the bits within a digit. These timing tracks may be used either as the pulse timer for the entire computer or for synchronizing the drum with the rest of the computer. Usually the read-write heads are not placed in line as shown in Fig. 6-3-1, but are staggered around the drum so that there will be a minimum of interference, physical and electromagnetic, between the heads. This spacing is irrelevant in either the use or internal operation of the computer.

For the 4,000-word drum described above, there are 200 words in each track. Using four digits for the address of a word, the first two may be used to designate the location of the half of the track containing the word and the second two digits for the word location within the half of the track.

The average access time (the time to locate a desired word from memory assuming the drum location is arbitrary at the start of the look-up) for the preceding magnetic drum is approximately one-half of a drum revolution or $\frac{1}{120}$ of a second, approximately 8.33 msec (milliseconds) or 8.33×10^{-3} second. Since it takes

$$\frac{1}{200} \times \frac{1}{60} \text{ sec} = 0.0833 \text{ msec}$$

to read one word from the drum and not more than eight word times to perform a CAU or HAU order on a single or two-address computer and since most of the orders require about this amount of time for execution, the major portion of the computation time of a drum computer may be consumed in waiting for words to appear under the read-write heads. The operation of a drum computer is usually improved; that is, the ratio of average access time to average computation time is decreased in one of two ways. The first is by the construction of a small, fast memory (frequently called a *buffer memory* as it may also be used in the transfer of blocks of computer words and as a timing buffer in input and output operations) in the computer, and the second is by use of coding techniques referred to as minimum access or optimum coding.

A buffer memory may be added to a drum memory by adding another track on the drum and forming a *delay line* as follows. Separate read and write heads are associated with this track, and, in the direction of rotation of the drum, the read head is displaced around the track from the write head by the number of words to be stored in the buffer memory. Thus a position on the track first appears under the write head and then, through the rotation of the drum, appears under the read head as shown in Fig. 6-3-2. The quotient of the number of words contained in a memory track divided by the number of words contained in a buffer

memory is usually chosen to be an integer. For a memory track containing 200 words, a buffer memory might contain 20 words. The operation of the read and write heads is as follows: The words desired in the buffer are entered through the write head. After the 20th word has been entered, the first word appears under the read head and by the cross-connection of the read and write heads, the 20 words will again be written in the next 20 locations of the track. By leaving the read and write heads cross-connected, a word in the buffer is read ten times for every revolution of the drum. This procedure reduces the average access time to approximately 0.833 millisecond or ten word times, a time which is of the same order of magnitude as that for an addition.

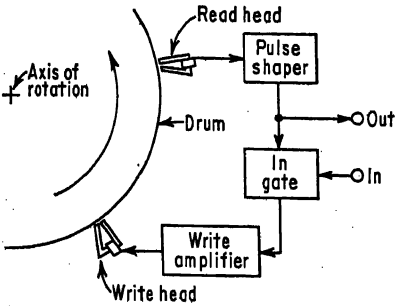


FIG. 6-3-2. A drum buffer memory.

Usually a computer utilizing a drum buffer memory as described has several buffers so that one or more may be used to hold instruction words and one or more may be used to hold information words. These buffer memories are also used for transferring blocks of words (in the case of the drum buffer described, a block consists of 20 words) from one part of the memory to another, for transferring blocks of words to memory from an input unit, and for transferring blocks of words from the memory to an output unit. Locations within the drum buffer memory are assigned addresses as are other memory locations. For the drum memory described, the addresses of the buffer locations of the first buffer memory might be 4,000 to 4,019, locations of the second buffer memory might be 5,000 to 5,019, etc. Instructions associated with the buffer memories include block transfer orders such as: (1) transfer the block of 20 consecutive words from memory, location of the first of these words being M, to buffer starting at buffer address N; and (2) block transfer from buffer, starting at address N, 20 words to consecutive memory locations, the location of the first word being memory location M. For most computers with drum buffer memories there is a restriction on the addresses M and N, i.e., M and N must have the same remainders when divided by the number of words in the buffer. This restriction is to ensure that the corresponding memory locations may appear simultaneously under the reading heads.

Another principle used in constructing a delay line is that of circulating compression waves through a medium. For example, an electri-

cal pulse is used to initiate a compression wave at one end of a column of mercury. This compression wave travels down the column of mercury to the other end where it is converted to an electrical pulse which is then used to gate a new electrical pulse initiating a repeat of the compression wave (thereby completing the delay repetition process). Since a compression wave travels 1 inch in mercury in approximately 17.5 microseconds, about 17 compression waves may be packed in every inch of mercury delay line assuming an electrical pulse length of 0.5 microsecond separated by 0.5 microsecond. Thus, ten words of 41 bits each may be packed into a mercury delay line which is approximately 24 inches long. A 1,000-word memory would consist of 100 such delay lines. A timing device would be associated with these delay lines to keep track of positions for compression waves, and the presence or absence of a compression wave at a given time would indicate whether the bit is a one or a zero. Using a three-decimal digit address for the location of a word in such a memory, the first two digits are used to designate the delay line, and the third digit to select the timed position of the word within the delay line. A new word is stored in the memory by using the bits of the word to gate the electrical pulses which initiate the compression wave in the proper delay line at the proper time (for the location of the word at the start of the mercury column) instead of using the pulses generated by the compression waves of the word previously stored. The average access time for a word (half of the time of travel of a word from one end to the other of a delay line) in such a delay line is approximately

$$12 \times 17.5\ \mu\text{sec} = 0.21\ \text{msec}$$

or five word times, which is of the same order of magnitude as the time for an addition operation in the existing computers of this type.

Delay lines, using the principle of compression waves, have also been constructed of a magnesium alloy and of fused quartz. These delay lines suffer from the fact that the speed of the compression wave depends upon the temperature of the medium, and the temperature of the medium must be carefully regulated to keep the compression waves in phase with the timing equipment. Delay lines are also used in the arithmetic and control units of the computer to delay pulses. Delay lines for these purposes are usually *lump constant* delay lines which are made of electronic components.

6-4. Minimum Access or Optimum Coding

Let us consider a drum memory, as described in the preceding section, without buffer tracks, for the main memory of a two-address computer

where the first address M_1 is an operand or data address, and the second address M_2 is the address of the next instruction. The object of optimum coding, for such a computer, is to minimize the waiting times for each instruction, for each operand called from memory, and for each result sent to memory. The waiting time is the time required for the desired memory location to rotate to a position under its read-write head.

In order to discuss optimum coding more easily, let us assign word locations on the surface of the drum. Let the drum have 20 tracks of 200 words each and assign the drum angle 0° to the drum position when the first word of each track is under its respective read-write head. The remaining word locations are placed every 1.8° around the track. Thus, when the drum is at 0°, word locations addressed $200p$, $0 \leq p \leq 19$, are under the read-write heads of the tracks numbered 0 through 19, respectively. When the drum has rotated through 1.8°, word locations addressed $200p + 1$ are under the read head, etc. Now, let us assume that an order word containing an addition operation is located at address M and that three word times after it is called from memory, the arithmetic unit is ready for the operand. Thus, the optimum location of M_1, the data address of the addend, would be $M + 3$. Assuming that it takes four more word times after reading the addend to complete the addition, the computer is ready to receive the next instruction from word location $M_1 + 5$, and the optimum instruction address within the order located at M should be

$$M_2 = M_1 + 5 = M + 8$$

Reflecting back on what we have already said, the address M_1 need not be $M + 3$, but might be any similarly located address of another track, say $M + 3 \pm 200p$; and, similarly, the address M_2 might also be $M_1 + 5 \pm 200k$ where p and k are any integers in the range $0 \leq p, k \leq 19$, for which

$$0 \leq M + 3 \pm 200p, \quad M_1 + 5 \pm 200k \leq 3{,}999 \qquad (6\text{-}4\text{-}1)$$

Another way of expressing the possible address assignments of M_1 and M_2 is acquired with the concept of congruent integers. If two integers s_1 and s_2, when divided by the integer d, have remainders r_1 and r_2 which are the same, i.e.,

$$r_1 = s_1 - dq_1 = s_2 - dq_2 = r_2$$

then s_1 and s_2 are said to be congruent modulo d; and this statement is written as

$$s_1 \equiv s_2 \pmod{d}$$

For example, all odd integers are congruent modulo 2,

$$1 \equiv 3 \equiv 5 \equiv 7 \equiv 9 \pmod 2 \equiv \cdots$$

since

$$\begin{aligned} 1 &= 1 - 2 \cdot 0 \\ 1 &= 3 - 2 \cdot 1 \\ 1 &= 5 - 2 \cdot 2 \\ 1 &= 7 - 2 \cdot 3 \qquad \text{etc.} \end{aligned}$$

Let $\{x \geq 0\}$ be the set of all nonnegative integers x such that

$$x \equiv s \pmod d$$

then

$$x = dq_x + r \qquad \text{and} \qquad s = dq_s + r$$

where q_x is an integer depending upon x; and

$$\min_{\{x \geq 0\}} [x \equiv s \pmod d] = r(s,d) \tag{6-4-2}$$

The integer $r(s,d)$ is called the remainder or *residue* of s modulo d and $0 \leq r(s,d) < d$.

Returning to the possible addresses of M_1 and M_2, let

$$r_1 = r(M + 3, \quad 200) \tag{6-4-3}$$

and

$$r_2 = r(M_1 + 5, \quad 200) \tag{6-4-4}$$

then

$$M_1 = r_1 + 200p \tag{6-4-5}$$

and

$$M_2 = r_2 + 200k \tag{6-4-6}$$

where $0 \leq p, k \leq 19$. Should none of the locations M_1 as defined by Eq. (6-4-5) be available, then the next best choice for M_1 would be

$$M_1 = 1 + r_1 + 200p$$

for some integer p in the increased set $-1 \leq p \leq 19$. In general, the best choice for M_1 is given by

$$M_1 = t + r_1 + 200p \tag{6-4-7}$$

where t is the smallest integer which for some p, $-1 \leq p \leq 19$, produces the address of an available location. Similarly, the best available location for M_2 is

$$M_2 = u + r_2 + 200k \tag{6-4-8}$$

It is probably apparent to the reader that optimum coding requires a table which for each order gives the number of word times d required by the computer to read and decode an order and the number of word times i to execute the operation. Such a table is often referred to as an *optimum coding chart* or *timing chart*, and an example of such a chart is given in

Table 6-4-1. In addition to having an optimum coding chart, the programmer will have to construct a memory assignment chart showing the memory location assignments as they are made during the process of optimum coding.

The abbreviations used in the optimum coding chart are those used for the two-address computer of Appendix II. For the operations MLR and MLT, S_m represents the sum of the digits of the multiplier; for the operations DVR and DIV, S_q represents the sum of the digits of the quotient; and for the operations SHR and SHL, n represents the number of digit positions the contents of the accumulator is shifted. Equations (6-4-3) and (6-4-4) may be extended to the operations of Table 6-4-1 and become

$$r_1 = r(M + d, \quad 200) \tag{6-4-9}$$

and

$$r_2 = r(M_1 + i, \quad 200) \tag{6-4-10}$$

TABLE 6-4-1. Optimum Coding Chart for Two-address Computer of Appendix II

Operation	d	i
CAU, HAU, CMU, HMU, CSU, HSU CAL, HAL, CML, HML, CSL, HSL	3	5
CAQ, CMQ, CSQ	3	3
MLR MLT	3 3	$25 + S_m \leq 115$ $20 + S_m \leq 110$
DVR DIV	3 3	$68 + S_q \leq 158$ $60 + S_q \leq 150$
STU, STL, STQ	5	3
TAP, TAN, TAZ, OVW	4	4 #
SHR, SHL	—	$10 + 2n$ #
XTR	5	5

By using Table 6-4-1 and Eqs. (6-4-9) and (6-4-7), the optimum address for M_1 of each order may be assigned; and by using the table and Eqs. (6-4-10) and (6-4-8), the optimum address M_2 (and the location of the next order word) may be assigned. A # symbol in column i means that M replaces M_1 in Eq. (6-4-10). In these cases M_1 is not a data address.

So far, we have discussed optimum programming as though there were

no interplay between orders, as though we had complete freedom in the location of data in the memory, and as though there were no restrictions on the scattering of orders throughout the memory. Before discussing the influence of these restraints on optimum coding, we should point out that if one knows S_q, the sum of the quotient digits, then one usually knows the quotient digits and the division operation was not necessary. Thus, i might be chosen to be $150 = 60 + 10 \times 9$ for the DIV operation and 158 for the DVR operation, which are the maximum values of i, for these operations. Since $60 \leq i \leq 150$ for the DIV operation and $68 \leq i \leq 158$ for the DVR operation, a more optimum choice of i in each case would be a value which minimizes the average waiting time for the expected distribution of the S_q's.

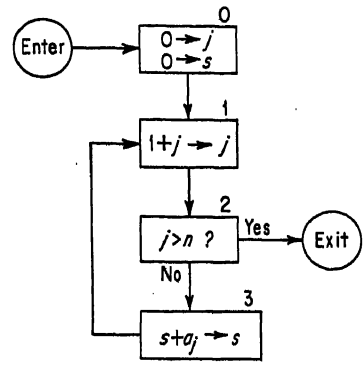


FIG. 6-4-1. Flow chart for $s = \sum_{j=1}^{n} a_j$.

Often the multiplier is a known constant, and i for the operations MLR and MLT may be determined. However, if the multiplier is a variable, determined by other calculations, then the choice of i might be such that $20 \leq i \leq 110$ for the MLT operation and such that $25 \leq i \leq 115$ for the MLR operation where the exact choice of i, in each of the two cases, would be the value which minimizes the average waiting time for the expected distribution of the S_m's.

In optimum programming one does not always have complete freedom in assigning locations for the data. For example, if we desire to obtain

$$s = \sum_{j=1}^{n} a_j$$

in a program, the a_j's would be stored in sequence so that a simple program as shown in Fig. 6-4-1 might be used. If this program is optimized with respect to a_1, then it will be less optimized with respect to a_2 and even less optimized with respect to a_3. For box 1 it is necessary to have the a_j in sequence so that the required modification of addresses can be performed which, of course, precludes optimizing box 3 for all j. Figure 6-4-1 also demonstrates the interplay between orders. Since the M_2 address of the last order for box 3 is fixed (it is the address of the first order in box 1), it will not, in general, be optimized.

An important use for optimum coding is in the construction of codes

for frequently used subroutines. Generally, subroutines calculate the value of dependent variable $y = f(x_1, x_2, \ldots, x_n)$ from the independent variables x_i, $i = 1, 2, \ldots, n$, where the number n of independent variables is small. An example of a subroutine is the calculation of $y = \sin x$ where x is the independent variable supplied to the routine. The requirements of a subroutine are that its computation time be as small as possible, that the number of memory locations required for the routine be as few as possible, and that these locations be as compact as possible. The requirement on computation time implies optimum coding insofar as is practicable. Minimizing the number of locations requires using the same orders over when iterative schemes are used. Compacting memory locations means that there should be a minimum of unused memory cells between the first and last locations used in the subroutine. The last requirement results from the use of several subroutines in programming a problem. If the words of a subroutine were scattered throughout the memory, then there would be the task of interlacing the memory locations of several subroutines which, in addition to being difficult, might not be attainable. The requirements for optimized subroutines are, of course, desirable for the coding of any program where optimum coding may be employed.

Another way to obtain an optimized program compiled mainly of subroutines is by having a subroutine library comprised of sequentially coded subroutines with indices that identify the relocatable addresses in the subroutine. The subroutines are incorporated in the program and then the entire program is optimized. Often a subroutine library for a computer which requires optimized coding will have both compact optimized and sequential subroutines. In the long run, programs comprised of the former type of subroutines require less optimizing effort since the major effort is made once during the construction of the subroutines, whereas programs comprised of the latter type of subroutines which are then optimized may obtain a higher degree of optimization since there usually are not as stringent requirements upon the utilization of memory space.

The programmer's extra effort involved in the optimum coding of a short problem is often more costly than the increased computer cost of running the problem without optimizing techniques. In these cases semioptimizing or machine optimizing techniques are often used. One type of semioptimizing technique may be developed by noting in the optimum coding chart of Table 6-4-1, that i and d are less than or equal to 5 except for the six operations where i is variable. By assigning in order those addresses which are to be optimized as $200p + r(5q,199)$ for $q = 0, 1, 2, \ldots, 198$ and $200p + q$ for $q = 199$ for a given p, memory locations

are filled in blocks of 200 locations. If $p = 0$, then $q = 0, 1, 2, \ldots, 39$ assigns locations addressed 0000, 0005, 0010, . . . , 0195; $q = 40, 41, 42, \ldots, 79$ assigns locations addressed 0001, 0006, 0011, . . . , 0196, etc. Thus, if the first-order word is located at 0000, its data address will be 0005, which will be the location of that data word, and its instruction address will be 0010 and the second order will be located there. If the data for the order word addressed 0000 was prelocated (or the data address was not to be optimized), then the instruction address would be 0005. The result of semioptimized coding is, except for the six orders already mentioned, that the waiting time for either semioptimized data or order words is never greater than three[1] word times. Two advantages of semioptimum coding are that it is easily performed and that it produces a reasonably fast and compact code.

The optimized coding previously discussed may be made into a computer routine.[2] For such a routine, one requires that a coded program for a problem has already been developed in some form. The optimizing routine is to produce from this original code an optimized code. Let the original coded program be contained in an input deck of cards where each card contains the following information for each word of the program:

1. L', the memory location of the word as used in the original code
2. Contents of the word at location L'. (If the contents represent an order, then the sign and the first two digits represent the operation, the next four digits represent the data address M'_1, and the last four digits represent the instruction address M'_2.)
3. A one-digit location index x
4. A one-digit data address index y
5. A one-digit instruction address index z

$$\text{If } x = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \quad \text{then } L' \begin{cases} \text{is fixed} \\ \text{is to be optimized} \end{cases}$$

$$\text{If } y = \begin{Bmatrix} 0 \\ 1 \\ 2 \end{Bmatrix} \quad \text{then } M'_1 \begin{cases} \text{is fixed} \\ \text{is to be optimized} \\ \text{is not an address} \end{cases}$$

$$\text{If } z = \begin{Bmatrix} 0 \\ 1 \\ 2 \end{Bmatrix} \quad \text{then } M'_2 \begin{cases} \text{is fixed} \\ \text{is to be optimized} \\ \text{is not an address} \end{cases}$$

[1] This maximum waiting time occurs for $d = 3$ or $i = 3$ when the pairs of semioptimized addresses differ by six locations; e.g., 0195 to 0001, 0196 to 0002, etc.

[2] The computer routine described here is essentially that of Barry Gorden, An Optimizing Program for the IBM 650, *J. Assoc. Computing Machinery*, vol. 3, pp. 3–5, 1956.

Each input card contains one word of the original program and the essential information for optimization of that word. For order words the information indicates whether the digits of the data address represent a data address or a subinstruction (as in a shift order).

The object of the optimizing routine is to generate a new deck of input cards in which L', M'_1, and M'_2 are replaced by their optimized values L, M_1, and M_2 as indicated by the indices. For this purpose, let the first 2,000 words of memory, $0000 \leq M \leq 1999$, be used as an *assignment table* for the optimizing routine, and let the memory locations 2,000 through 3,999 be used for the orders of the optimizing routine. The assignment table is used in the following way: The first four digits, d_0, d_1, d_2, and d_3 in the contents of location M of the assignment table designate the new address in the optimized program for a word in the original program, the address M' of which was equal to M, $0000 \leq M' = M \leq 1999$; and the four digits d_5, d_6, d_7, and d_8 in location M of the assignment table are the new address in the optimized program for the word the previous address M' of which had the value $M' = M + 2000$, $2000 \leq M' \leq 3999$. If the digit d_4 in the location M in the assignment table equals 1, the location M has been used in compiling the optimized program; and, if $d_4 = 0$, the location M has not been used. If d_9 in the location M in the assignment table equals 1, the location $M + 2000$ has been used in generating the optimized program; and, if $d_9 = 0$, the location $M + 2000$ has not been used. This table, then, provides the optimizing routine with the optimized address assignments for the words of the original program and an indication of the memory locations which have been used by or are available for the optimizing routine.

The operation of the optimizing routine is as follows: First, the original input deck is read into the computer and the optimizing routine records in the assignment table all fixed addresses. That is, if L', $0000 \leq L' \leq 1999$, is fixed ($x = 0$), then the digits d_0, d_1, d_2, and d_3 of location $M = L'$ of the assignment table are set equal to L', and the digit d_4 is set equal to 1; or if $2000 \leq L' \leq 3999$, then the digits d_5, d_6, d_7, and d_8 of location $M = L' - 2000$ of the assignment table are set equal to L' and the digit d_9 is set equal to 1. Similarly, the fixed addresses M'_1 and M'_2 are recorded in the assignment table. After all fixed addresses have been recorded, the original input deck of cards is again read into the computer. After each card is read into the computer, the optimizing subroutine checks with the assignment table to see if L' has already been given an optimized address; and, if not, L' is optimized and so recorded in the assignment table. The contents of the word addressed L are then checked by the routine to see if the addresses M'_1 and M'_2 are to be optimized and, if so, M_1 and M_2 are recorded in the assignment table. Next, the optimizing

routine punches a new card containing L and its contents and calls for the next card of the original input deck. If M_1' and M_2' are to be optimized and have not been given previous assignments, M_1 and M_2 are calculated by Eqs. (6-4-7), (6-4-8), (6-4-9), and (6-4-10) where d and i are determined by having the optimum coding chart stored in the memory. If L has not been given a previous assignment, then L is determined by

$$L = \min_{p} [L' + p]$$

where p is a positive integer, which produces a previously unassigned location. The location 0000 (zeros in the assignment table are used to indicate that no assignment has been made) is prohibited in the optimized program by the subroutine. With an optimizing subroutine such as that described above, the presetting of the digits d_4 and d_9 may be used to restrict the optimized program to or from specified locations of the drum. These special locations could be reserved for subroutines or tabular data requiring sequential locations.

Some drum memories may have the angular position of words within tracks assigned according to the desires of the programmer. For example, first address may be assigned the first angular position within each track, the second might be assigned the seventh angular position, etc. These positions, then, may be assigned addresses so that serial addressing of words of a routine produces a semioptimized code. Consider a computer, the execution time of which is shown in the chart of Table 6-4-1, with a 2,000-word memory having 40 tracks with each track containing 50 words. If each of the sequential angular positions L within track t has the address A, the relation

$$L = r[7(A - 50t), 50] \tag{6-4-11}$$

for $50t \leq A < 50(t + 1)$, produces a semioptimized code when the words of the program are serially addressed. The maximum waiting time for an optimized data word or the next optimized order word, except for the six orders in which i is a variable, is four word times. From Eq. (6-4-11) we may see that

$$A(L, t + 1) = 50 + A(L,t)$$

that is, the address for angular position L of track $t + 1$ is 50 greater than that for L in track t, and a single address track suffices for the entire drum. Figure 6-4-2 shows the addresses for track positions on track $t = 0$.

Additional information on optimization procedures will be found in J. W. Carr III, *Handbook of Automation Computation and Control*, vol. 2, chap. 2, John Wiley & Sons, Inc., New York, 1959.

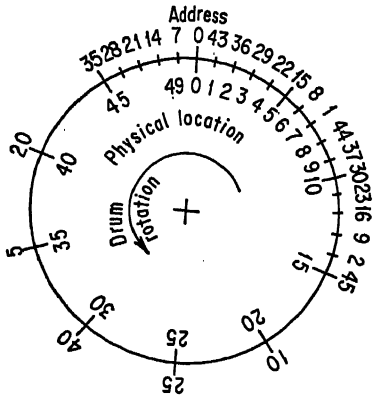


FIG. 6-4-2. Address assignment for a semioptimized drum memory.

PROBLEMS

6-1. Using the optimum coding chart of Table 6-4-1 and the orders for the two-address computer of Appendix II, construct an optimum code and memory chart for Prob. 3-2.

6-2. Often in taking statistical data, several variables are measured for each of a number of elements. For example, one might be measuring the height x_i and the weight y_i of the ith person of a group of N people, and one might want to know if there is a correlation between these variables. A measure of this correlation is given by the *simple correlation coefficient* r, where

$$r = \frac{s_{xy}}{s_x s_y}$$

The covariance s_{xy} is given by

$$s_{xy} = \frac{1}{N-1} \sum_{1}^{N} (x_i - \bar{x})(y_i - \bar{y})$$

s_x^2 is defined in Prob. 1-6, $s_x = |\sqrt{s_x^2}|$, and $\bar{x}$ is defined in Prob. 1-4. For $N \leq 300$, draw a detailed flow diagram which will allow one to obtain any of the quantities $\bar{x}$, s_x^2, s_{xy}, and r. That is, the routine for obtaining r should be equally usable for one desiring only $\bar{x}$, or $\bar{x}$ and s_x^2, or $\bar{x}$ and $\bar{y}$, or $\bar{x}$, $\bar{y}$, s_x^2 and s_y^2.

6-3. Using the three-address code of Appendix I and floating point arithmetic operations, construct a code for the flow diagram of Prob. 6-2. Assume the x_i and y_i are in floating point notation.

6-4. Using the two-address code of Appendix II, floating point arithmetic operations, and the optimum coding chart of Table 6-4-1 where floating point operations are assumed to have the same timing as their corresponding fixed point equivalents, construct an optimum code for the computation of s_{xy} as described in the flow diagram of Prob. 6-2.

7

MAGNETIC-TAPE UNITS AND PROGRAMMING

7-1. Introduction

Magnetic-tape units may be used for auxiliary memories and for input and output units. The storage capacity of magnetic-tape units is usually at least an order of magnitude larger than that of the main memory. Tape units are often used in applications involving large amounts of data or intermediate results. They are also used for storage of frequently used programs and subroutines. The magnetic-tape unit may be considered to be composed of a magnetic tape, a read-write head, two tape reels or baskets, and a control unit. The magnetic tapes are usually made of either a plastic or steel base with a magnetizable surface. They are approximately 0.002 inch thick and vary in width, depending upon the tape unit, from ½ to 2 inches. Tapes vary in length, depending upon the computer and the purpose of computation. The control unit, responding to commands from the computer, controls the motion of the tape, the reading and writing of information on the tape, and the transfer of information between the computer and the tape unit. The tape passes, from a reel or a basket, by the read-write head, which is capable of reading or writing bits on each track (*cf.* Fig. 7-1-1) of the tape, to another reel or basket. The reels or baskets of tape are usually replaceable so that the tapes can conveniently be used for temporary or permanent files. The linear speed for a tape is of the order of 100 inches per second, and the density of magnetic pulses representing bits within a track is of the order of 100 pulses per inch. Since the word time for a tape may be considerably longer than the computer word time, the information transferred between tape and computer generally passes through a buffer storage register capable of accepting information at either the computer or tape rate.

The manner in which information is recorded on a tape is to a large extent determined by the mode of computer operation. For example, consider a serial-parallel computer for which the words are composed of a sign digit and ten binary-coded decimal digits and where the sign and decimal digits are operated upon serially and the four coded bits for each digit are operated upon in parallel. A word on the tape may be recorded in four tracks with the bits of each digit recorded, one bit per track, in a row across the tape. For example, the word +0123456789 is shown in Fig. 7-1-1 as it might be recorded on a tape. The fifth track is for check bits which are formed as follows: Let

$$d = b_3 2^3 + b_2 2^2 + b_1 2 + b_0$$

where d is the decimal digit and b_i is the bit in the $(i + 1)$th track. The check bit b_4 is zero if the sum $b_0 + b_1 + b_2 + b_3$ is odd, and b_4 is 1 if this sum is even. That is, the fifth bit is determined so that the sum of the bits across the tape is odd. The reliability of the information processed by tape units is not as high as that processed by other computer units and checks on this information, such as the parity check bit described above, are incorporated so that a large percentage of tape reading errors will be detected.[1] There are also some coding systems that use the redundant bits to correct errors. One of these systems uses check bits in each row and in each column of a block of information on the tape. These bits are used to locate the row and column of the incorrectly read bit. In many of the present tape units, with exposed magnetic material, the read-write heads collect magnetic oxide dust from the tapes, causing errors in reading of the digits on the tape.

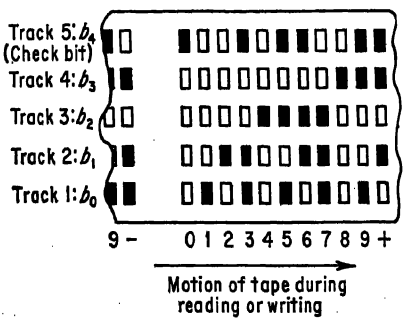


FIG. 7-1-1. Digit recording on a magnetic tape.

The speed of the magnetic-tape unit is slower than that of the main memory and faster than that of card or paper-tape input-output units. Also, for a given amount of information, tape files are less bulky than paper-tape or card files. Tapes have another advantage in being reusable.

The order of storing the digits on the tape shown in Fig. 7-1-1 is the same order in which they are transmitted between units of a serial-

[1] For other checking schemes, see R. K. Richard, "Arithmetic Operations in Digital Computers," pp. 187–192, D. Van Nostrand Company, Inc., Princeton, N.J., 1955.

parallel computer having a single decimal digit adder and a one-word buffer. The first digit read of each word on the tape is the sign digit. If the sign digit of the word is 1010, the word is positive; and if the sign digit is 1011, the word is negative. The remaining digits are read in the order from least significant (9 in Fig. 7-1-1) to most significant (0 in Fig. 7-1-1).

In Fig. 7-1-1, a space equal to that required for recording two characters has been left between the words. This space is equivalent to a time which is greater than or equal to the time used to transfer the word from the tape buffer to the computer.

Let us consider a simple procedure for the transfer of information from a magnetic tape, as in Fig. 7-1-1, to a magnetic-drum memory of a serial-parallel computer. For the tape unit, let the tape speed be 100 inches per second and the pulse density be 60 pulses per inch. The time t_T required to read one word (11 characters) from tape is $11/(60 \times 100)$ second or $t_T = 1.833 \ldots$ milliseconds; and the time t_B between words is $2/(60 \times 100)$ second or $t_B = 0.33 \ldots$ millisecond; i.e., two tape digit times. Let the drum memory have main memory tracks containing 200 words each and at least one buffer memory track of 20 words, and let the drum speed be 3,600 rpm; then the time t_C required to read one word from the drum memory, that is the word time within the computer, is $1/(60 \times 200)$ second or $t_C = 0.0833 \ldots$ millisecond. The longest waiting time for access to a desired location in the 20-word drum buffer memory (DBM) is 20 t_C or 1.66 . . . milliseconds. By adding a one-word tape buffer register (TBR) and by assuming (1) that the contents of TBR may be transferred to the upper accumulator A_U in two computer word times and (2) that one computer word time is required to prepare the computer for the transfer of the contents of A_U to DBM, 20 words may be transferred under continuous operation from the tape to the computer, filling the DBM. During the time t_B, the contents of TBR are transferred to A_U; and during the time t_T, a word is transferred from tape to TBR and the contents of A_U are stored in DBM. A timing chart for this operation is shown in Fig. 7-1-2. The transfer from A_U to DBM shown in this chart is the longest waiting time, i.e., 20 t_C.

The words may be placed on the tape in groups of 20 words called *blocks*. The contents of one block fill the DBM. By leaving sufficient space between blocks, time is available for the transfer of the contents of the DBM to the main memory before the next block is ready to be read into the DBM. The maximum waiting time for access to the first of twenty desired sequential locations in main memory is 200 computer word times, and the transfer time for the contents of DBM to twenty

sequential locations in main memory is 20 computer word times. If one computer word time is required to prepare the computer for the transfer of the contents of DBM to main memory, the maximum time required to transfer the contents of DBM to a desired set of 20 sequential locations in main memory is 220 computer word times or 18.33 . . . milliseconds. By having a space between blocks on the tape equivalent to a time greater than or equal to 18.33 . . . milliseconds (1.833 . . . inches or

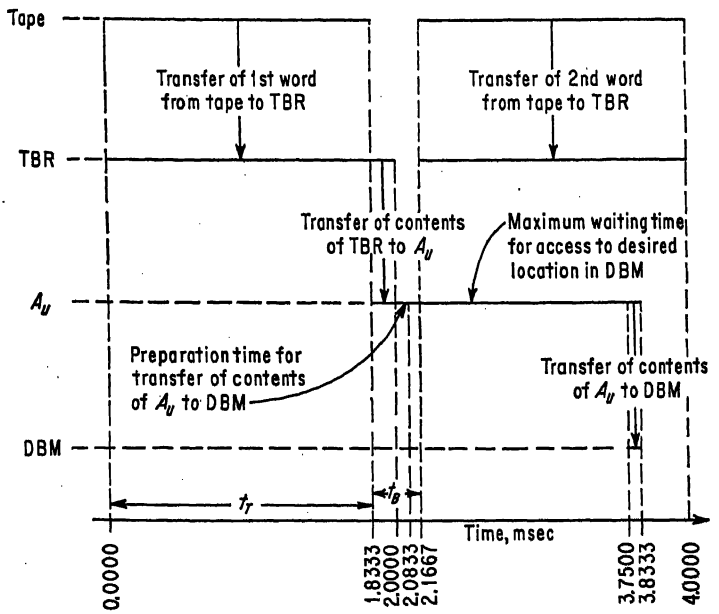


FIG. 7-1-2. Timing chart for the transfer of information from magnetic tape to drum memory.

the length for one tape word), blocks of words on the tape may be transferred to sequential locations in the main memory while the tape is in constant motion.

Such a detailed description as the one just presented, in addition to exhibiting the synchronizing of the operation of the tape unit with the computer, shows which parts of the computer are involved in the transfer of information between computer and tape unit. For example, one sees that information stored in the DBM and the previous contents of A_U will be destroyed by the execution of a command causing the transfer of information between tape unit and computer.

The preceding description is only one of many ways of transferring data between tape and main memory. The controlling factors in determining the method of transfer are the type of main memory, the mode of computer operation, and the structure of the buffering system. Another major factor is the contemplated use of the computer. These factors not only control the manner in which the information is transferred between tape and main memory, but are also major factors in determining the manner in which the information is recorded on the tape. For example, the bits of each word might be recorded in a row across a tape having as many tracks as there are bits in the computer word. Such a tape might be used with a computer having a parallel mode of operation or having a large buffer memory. The TBR might be replaced by a large tape buffer containing as many words as there are words in a block on the tape. The advantage of such a buffer is that the computer may be used for calculation purposes during the time the tape buffer is being loaded. If the cells of the buffer are addressable, the buffer could also be used for additional computation storage.

Some tape units use tapes that have addressable blocks. For such tapes the address of the blocks may be inserted in the space between blocks or on an additional track or tracks. If the address portion of an order word is used to refer to block addresses, then, for a 4-decimal-digit address, the maximum capacity of a tape is 10,000 blocks or 200,000 words if each block contains 20 words. Other tape units use tapes that do not have addressable blocks. Part of coding and programming problems using such tapes involves keeping track of where the tape is and where the information is located on the tapes so that the appropriate commands can be used to move the tape to the proper position for reading the information desired.

For a magnetic tape which is 2,000 feet long, the time required to pass the entire tape under the reading head is 4 minutes (assuming a tape speed of 100 inches per second). In the computer processing of data from two widely separated blocks on the tape, it is desirable not to have computation stop while the tape is moving from one block to the other. The time the computer is idle for the purpose of bringing a desired block on a tape into position for transferring information to or from the memory may be reduced by designing the tape control unit to locate a desired block on the tape independent of the computer. That is, after the computer has indicated the desired block to the tape control unit (or the direction and number of blocks to be moved by the tape control unit), the tape unit and the computer are logically disconnected, and the computer may be used to perform other orders not involving that tape unit

while the tape unit is bringing the desired block into position. When the desired block is brought into position by the tape unit, the tape is stopped, and the tape unit may then receive another instruction, that is, it waits until a command with the proper tape unit number is encountered. Since most tape units must run at a specified speed for reading and writing operations, the position at which a tape is stopped must be properly spaced with respect to the read-write head so that when information is to be transferred between the memory and the tape, the tape will be brought to proper speed. A possible way to obtain the proper rest positions for the tape (positions at which the tape may be stopped) may be accomplished by inserting a *block mark* in the space between blocks. The block mark may be a special configuration of bits in the tracks used for digits or a bit on an additional track. The block mark serves two purposes. The tape is stopped at a sufficient distance from the mark so that when started the tape will be at the proper read-write speed when it passes the block mark. The block mark may also be used logically as a switch so that information is not transferred to or from the tape until the read-write head has passed a block mark.

Two possible orders for computers having tape units are:

1. Read from tape, of tape unit M, n blocks to sequential locations in the main memory with the first word written in the memory cell addressed L.
2. Write on tape, of tape unit M, n blocks of words from sequential locations in the main memory with the first word read from address L.

If the tapes have addressable blocks, then the computer may have a search order of the form:

3. Search for block addressed N on the tape of tape unit M.

For this order, the searching by the tape unit should be independent of the computer once the command has been decoded. Should tape unit M already be searching for a block designated by a previous command, a new search command will interrupt the procedure and replace the previous block address by the new block address. Thus, search commands may be corrected during a tape search. However, a read or write command will stop computation until the completion of a search command if one is in process.

If the tapes do not have addressable blocks, then the computer might have two orders (often referred to as *jump*, *move*, or *search* orders) of the

following form:

4. Move the tape of tape unit M forward p blocks.
5. Move the tape of tape unit M backward p blocks.

For these orders the motion of the tape should be independent of the computer once the command has been decoded. If the tape of such a tape unit has already been set in motion by one of the orders, 4 or 5, that order must be completed before another tape order is performed by the same unit.

Almost all computers have a rewind order for tape units; i.e.,

6. Rewind tape on tape unit M.

This order returns the tape on the specified tape unit to the first block on the tape. The rewind is at a higher speed than a search operation.

Tapes with addressable blocks usually allow the replacing of a block of information with new information. In some cases for nonaddressable tapes, the new information can only be inserted by rewriting the entire tape. That is, if the last block of information on a nonaddressable tape is the nth block, then the $(n + 1)$th, $(n + 2)$th, . . . blocks may be added in sequence, but mth block, $1 \leq m \leq n$, may not be replaced with new information without rewriting the entire tape.

With regard to the computers of Appendices I and II, except for rewind commands, the command list already contains a minimum set of commands that could be used for operating tape units. For example, consider the three-address computer of Appendix I, and let there be n tape units, addressed $M_1^{(1)}$, $M_1^{(2)}$, $M_1^{(3)}$, . . . , $M_1^{(n)}$, associated with the computer. The order SWL $M_1M_2M_3$ or SWP $M_1M_2M_3$ is used to activate the read-write connection between the ith, $1 \leq i \leq n$, tape unit and the computer when the address $M_1 = M_1^{(i)}$. By assuming that information is recorded on the tapes in nonaddressable blocks of 100 words per block, the order LOD $M_1M_2M_3$ is used to transfer information from the tape unit (connected to the computer) to the main memory. For the LOD order, M_1 designates the number of blocks to be loaded. That is, the digit d_1, $0 \leq d_1 \leq 9$, of M_1 designates the number of blocks and the digits d_2 and d_3 are ignored by the tape control unit. The address M_2 designates the address of the memory cell into which the first word of the first block is entered. The remaining words of the blocks are loaded into sequential memory cells. Similarly, the PRT command is used to transfer information from main memory to the tape unit con-

TABLE 7-1-1. Rewind Order

Symbol	Operation code		M_1 M_2 M_3	Explanation
	s	d_0		
REW	6	1	M_1 — M_3	Rewind tape on tape unit addressed M_1 and change CC to M_3. Ignore M_2.

nected to the computer. Both the LOD and PRT orders are unidirectional when used as tape commands; i.e., if the $(n-1)$th block has been read (written), then the nth block must be read (written) next unless the tape is rewound, in which case the first block must be read (written) next. The rewind order REW rewinds the tape of a designated tape unit whether or not the read-write connection to the computer is activated. The REW order is described in Table 7-1-1.

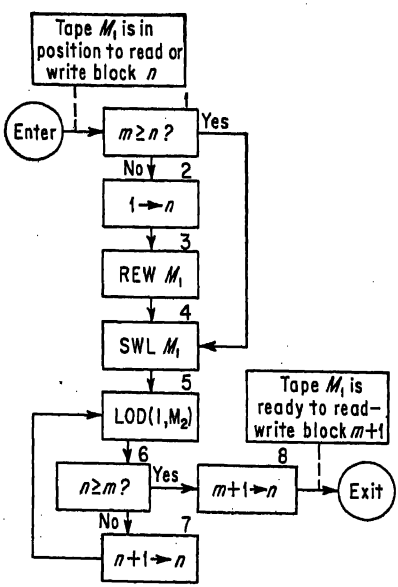


FIG. 7-1-3. Routine for loading a desired block from a tape into the memory of the computer of Appendix I.

Many problems requiring tape units do not use the information sequentially, block by block, from the tapes. Thus, when using the commands of Appendix I, where the blocks are not addressed and where search commands are not included in the command repertoire, the routine for the problem must keep track of the next block the tape unit can read and determine how to read the next desired block. For example, if during the computation of a problem the mth block of words on a tape is required and that tape unit is prepared to read the nth block, where n need not equal m, then the problem routine must determine a procedure for going from the nth to the mth

block. A possible flow diagram for such a routine is given in Fig. 7-1-3. In constructing this flow diagram it was assumed that the mth block from the tape of tape unit M_1 is to be read into memory cells addressed M_2 through $M_2 + 99$ and that the tape is in the position for reading the nth block. Box 1 of the flow diagram determines whether or not the tape must be rewound before the mth block is read. If $m < n$, then, in boxes 2 and 3, n is set to one, and the tape is rewound. Each time the routine passes through box 5, one block is read into the memory cells reserved for the mth block [LOD(M_1,M_2) represents the command LOD $M_1M_2M_3$]. Thus, each time a block is read into memory cells M_2 through $M_2 + 99$, it replaces the previous contents of those cells; and when $n = m$, the mth block is read into the desired memory cells. In box 8, the new position of the tape of tape unit M_1 is recorded. This information is indicated on the flow diagram in the unnumbered assertion box. The dashed line connecting the assertion box to the flow chart indicates where the assertion is valid.

7-2. Use of Magnetic Tapes as an Auxiliary Memory

Of all the problems involving the use of tapes, the simplest type to code for a computer are problems for the computation of $y_i = f(x_i)$, $1 \leq i \leq N$, where N exceeds the capacity of the main memory of the computer. If the capacity of the main memory is M words and the routine for computing y_i involves $P < M$ words, then $M - P$ memory cells are available for the x_i's or the y_i's. Let B be the number of words in a block of words on a tape (assume that all blocks are of the same size and that $B < M - P$) and let K be the largest integer such that $KB \leq M - P$, then KB is the number of main memory locations reserved for holding either the x_i's or the y_i's. Let us assume that the x_i's are stored on tape 1 and, after calculation of the y_i's, the y_i's are to be stored on tape 2. A procedure for computing the y_i's is to bring K blocks from tape 1 into the main memory; to calculate for each number x_i in the main memory the corresponding value for y_i; and to replace each number x_i in the main memory by the number $y_i = f(x_i)$. After these KB values y_i have been calculated, they are transferred to K blocks on tape 2. The next K blocks of numbers x_i are brought into main memory from tape 1, and their corresponding values y_i are calculated and transferred to tape 2. The procedure is repeated until all y_i's have been calculated and transferred to tape 2.

The preceding example displays few of the problems encountered in the programming of routines in which tapes are used as an auxiliary memory.

In that example, about the only excuse for using tapes as an auxiliary memory occurs when the computation of $y_i = f(x_i)$ is a part of a larger problem. Otherwise, from 1 to $(M - P)$ of the numbers x_i might be loaded into the main memory from an input unit and the corresponding values y_i delivered to an output unit.[1] This process would be repeated until the N values y_i are recorded at the output unit.

For the next programming example, in which four tape units are used, let us assume that the tape units and the computer have the following properties. First, any block of words on a tape may contain an arbitrary number of words but must not exceed 100 words of information. Second, associated with each block of information the tape contains a one-word block-length indicator which may be used to designate the number of words of information contained in that block. Third, there is a one-word tape register T_R in the main computer associated with the block indicator. Fourth, the blocks on the tapes are not addressed. Last, the commands associated with these tape units are:

1. Move the tape of tape unit M forward p blocks.
2. Move the tape of tape unit M backward p blocks.
3. Rewind the tape on tape unit M.
4. Read one block from the tape of tape unit M into sequential locations of the main memory with the first word having address L. Read block-length indicator into T_R, the one-word tape register.
5. Write one block of length t_R (contents of T_R) words and the block indicator t_R on the tape of tape unit M from sequential locations of the main memory addressed L through $L + t_R - 1$ and T_R.
6. Transfer the contents of A_U to T_R.
7. Transfer the contents of T_R to A_U.

Whenever a block of information is written on a tape, the contents of T_R is transferred to the block indicator; and whenever a block of information is read from a tape, the block indicator is transferred to T_R. Reading or writing of a block leaves the tape in position to read or write the next block.

The problem we wish to program is a sorting process which consists of

[1] For some computer installations there is a speed advantage in using magnetic-tape units for computer input or output. These installations have independent devices for transcribing data onto magnetic tape and devices for transferring results from magnetic tape to a printer. Such equipment is referred to as *off-line* or *auxiliary* equipment.

a sequence of successive mergings of two sets of numbers and is known as a *merge sort* or as the Bucharest sort. In this process a set of N numbers to be sorted in ascending order is divided into two sets for which the number of elements differ by, at most, one. These sets are merged to form two new sets which are composed of partially ordered subsets. These two partially ordered sets are merged to form two sets composed of longer partially ordered subsets. The length of the partial orderings increases by a factor of at least 2 with each merge. The merging process is stopped when the merge produces only one ordered sequence. At most $\log_2 N$ merges are required. Let the original sequence of numbers be $\{x_n\}$ where $|x_n| < X$ for all n, $1 \leq n \leq N$. It is desired to arrange the numbers of this sequence so that they form a monotonic nondecreasing sequence of numbers $\{\bar{x}_m\}$; i.e., a sequence where $\bar{x}_{m+1} \geq \bar{x}_m$ for all m, $1 \leq m \leq N-1$. For the first merge, let the sequence $\{x_n\}$ be divided into two sequences $\{a_i\}$ and $\{b_j\}$. The elements a_i are the x_n for which $1 \leq i = n \leq t$ where t is the largest integer such that $t \leq N/2$ and $a_{t+1} = -(X+1)$, and the elements b_j are the x_n for $N/2 < n = j + t \leq N$ and $b_{N+1-t} = -(X+1)$. The marker element $-(X+1)$ has been added at the end of each sequence $\{a_i\}$ and $\{b_j\}$ to indicate the end of the sequence. The Bucharest sort is next described in detail starting with the general merging process. A heuristic description follows the detailed description. Subsequences $\{c_{1k}\}$, $\{c_{2k}\}$, . . . , $\{c_{sk}\}$, . . . , where $\{c_{sk}\}$ represents the ordered sequence $c_{s1} \leq c_{s2} \leq \cdots \leq c_{sk} \leq \cdots$, are obtained from the two preceding sequences $a_1, a_2, \ldots, a_i, \ldots, a_t$ and $b_1, b_2, \ldots, b_j, \ldots, b_{N-t}$ in the following manner.

The element c_{sk} is a_i if $a_i \leq b_j$ and $a_i \geq c_{s,k-1}$, and i and k are increased by one. If, however, $a_i \leq b_j$ and $a_i < c_{s,k-1}$, but $b_j \geq c_{s,k-1}$, then $c_{sk} = b_j$ and j and k are increased by one. If $a_i > b_j$ and $b_j \geq c_{s,k-1}$, then $c_{sk} = b_j$ and j and k are increased by one. If $a_i > b_j$ and $b_j < c_{s,k-1}$, but $a_i \geq c_{s,k-1}$, then $c_{sk} = a_i$ and i and k are increased by one. Finally, if $a_i < c_{s,k-1}$ and $b_j < c_{s,k-1}$, the subsequence $\{c_{sk}\}$ is completed; then, for the next sequence k is reset to 1 and s is increased by one, and if $a_i \leq b_j$, then $c_{s+1,1} = a_i$ and i and k are increased by one. However, if $a_i > b_j$ then $c_{s+1,1} = b_j$ and j and k are increased by one. The newly merged sequences $\{c_{sk}\}$ are divided into two sets of sequences as follows:

$$\{c_{1k}\}, \{c_{3k}\}, \{c_{5k}\}, \{c_{7k}\}, \ldots$$

and

$$\{c_{2k}\}, \{c_{4k}\}, \{c_{6k}\}, \{c_{8k}\}, \ldots$$

which are redesignated the sequences $\{a_i\}$ and $\{b_j\}$ for repeating the general merging process just described. By having the element $-(X+1)$,

which is less than any of the numbers to be sorted, at the end of each sequence $\{a_i\}$ and $\{b_j\}$, the sorting procedure, defined by these rules, will automatically form the ordered subsequences $\{c_{sk}\}$, from one sequence when the other has been exhausted.

A simple explanation for the sorting by the preceding process follows. Consider each number to be on a card. These cards are divided into two piles. Ascending sequences are formed by taking the numbers from the top of each pile in ascending order. When an ascending sequence stops, all of the cards for that sequence are alternately put face down in one of two output piles. After the input piles are depleted, the output piles are then turned face up and used as new input piles. This process

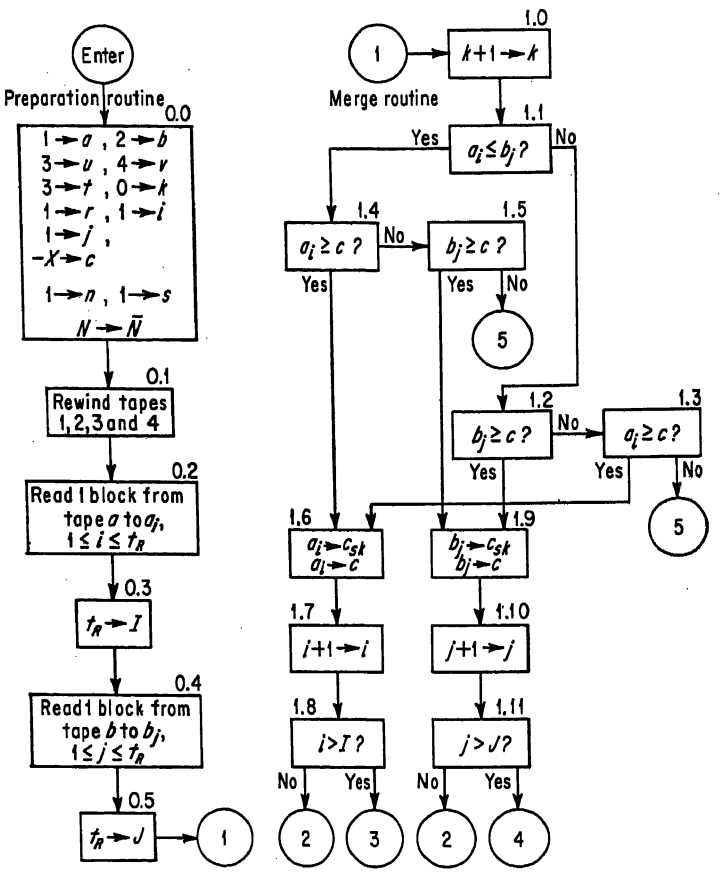


FIG. 7-2-1. Flow diagram for the Bucharest sort.

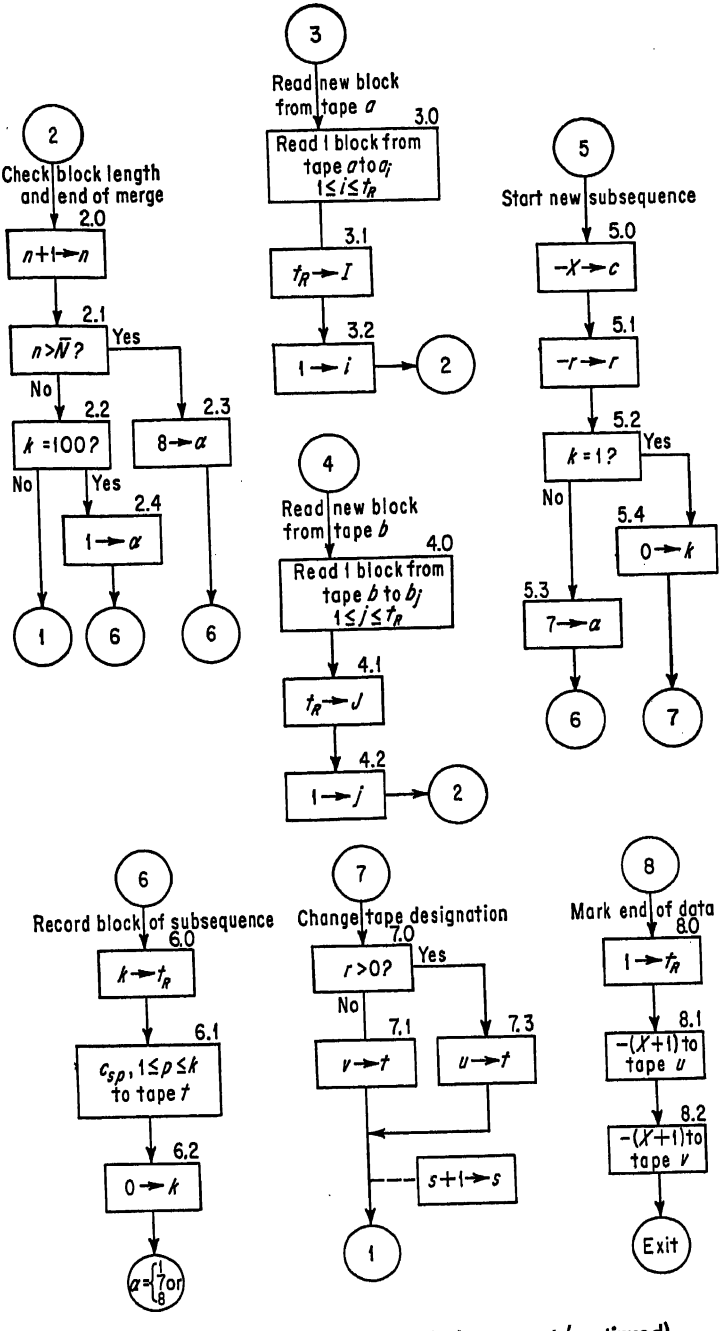


FIG. 7-2-1. Flow diagram for the Bucharest sort (continued).

is repeated until the numbers on the cards form one ascending sequence · i.e., are all in one output pile.

For the purpose of flow diagramming the Bucharest sort, let us designate the two new sequences that are formed from the sequences $\{c_{sk}\}$ by $\{u_l\}$ and $\{v_m\}$. The sequence $\{u_l\}$ is comprised of all subsequences $\{c_{sk}\}$ where s is odd and the sequence $\{v_m\}$ is comprised of all subsequences $\{c_{sk}\}$ where s is even. The sequences $\{u_l\}$ and $\{v_m\}$ are then terminated by $-(X+1)$ and are redesignated as the sequences $\{a_i\}$ and $\{b_j\}$, and the next new sequences $\{u_l\}$ and $\{v_m\}$ are formed. This procedure is continued until one of the two sequences $\{u_l\}$ or $\{v_m\}$ is empty, at which time the other sequence contains all of the numbers in a monotonic nondecreasing sequence.

The flow diagram of Fig. 7-2-1 is for compiling the sequences $\{u_l\}$ and $\{v_m\}$ from the sequences $\{a_i\}$ and $\{b_j\}$. The sequences $\{a_i\}$ and $\{b_j\}$ are on two input tapes and the sequences $\{u_l\}$ and $\{v_m\}$ are written on two output tapes. The formation of the two sequences $\{u_l\}$ and $\{v_m\}$ is completed when N terms c_{sk} have been merged. The programming is made more complex than the preceding description since a block of words on the tape is limited to a maximum of 100 words. Further, one must continually check to see that the main memory is supplied with a_i's and b_j's to be merged. Also, should a subsequence $\{c_{sk}\}$ have more than 100 terms, then that subsequence must be stored in blocks which do not exceed 100 words each. Initially, tape 1 contains the sequence $\{a_i\}$ and tape 2 contains the sequence $\{b_j\}$.

On output tapes 3 and 4, the sequence $\{u_l\}$ and $\{v_m\}$ will be composed of blocks, the sizes of which correspond to the length of the subsequences $\{c_{sk}\}$. That is, if the subsequence contains less than 100 terms, its block length will be the number of terms in the sequence. If $\{c_{sk}\}$ contains $K > 100$ terms, it will be stored in h blocks of 100 words each, where h is the largest integer such that $100h < K$, and one block of $K - 100h$ words.

The heart of the sorting routine is subroutine 1 which starts at the entry connector 1. This subroutine is constructed so that it follows the detailed description of the merging process as previously described. The number denoted by c in the comparisons of routine 1 is either $c_{s,k-1}$ or is $-X$, a value chosen to effect a meaningful comparison at the start for a new sequence. The exit connectors 5 indicate the completion of a subsequence $\{c_{sk}\}$ and the starting of a new subsequence $\{c_{s+1,k}\}$. Subroutines 5 and 7 cause the subsequences $\{c_{sk}\}$ to alternately become members of the sequences $\{u_l\}$ and $\{v_m\}$ and prepares for the start of a new output sequence. The subroutines 3 and 4 are used to refill the main memory

with a_i's and b_j's respectively. Subroutine 2 is used to transfer a block of c_{sk}'s to the proper output tape should the subsequence $\{c_{sk}\}$ exceed 100 words. In the flow diagram, k represents the number of the term within a tape block. Subroutine 6 is used to transfer the sequence $\{c_{sk}\}$ or parts thereof to the proper tape, u or v. The action is such that if more than one block is necessary for an ordered subsequence $\{c_{sk}\}$, all of that subsequence will be in only one of the two sequences $\{u_l\}$ or $\{v_m\}$. Subroutine 2 is also used to determine when all items of $\{a_i\}$ and $\{b_j\}$ have been sorted. Subroutine 0 is the preparation subroutine necessary for starting the sorting routine.

After the sequences $\{u_l\}$ and $\{v_m\}$ have been compiled, the merge process is completed by adding the word $-(X + 1)$ as the last member of each sequence $\{u_l\}$ and $\{v_m\}$, as done in subroutine 8. For the next merge, the tapes on tape units 3 and 4 are interchanged, either logically (i.e., by renumbering) or physically, with the tapes on tape units 1 and 2. The subroutine for starting the next merge by renumbering the tapes would differ from the preparation routine of Fig. 7-2-1 only in the tape designations of box 0.0. Under the worst conditions the original sequences $\{a_i\}$ and $\{b_j\}$ will contain only one-word subsequences, and if the worst conditions prevail, the first set of $\{u_l\}$ and $\{v_m\}$ will contain ordered subsequences each of which have two words, except for possibly one subsequence of one word. Continuing under worst conditions, the pth set of sequences $\{u_l\}$ and $\{v_m\}$ will contain subsequences each of which have 2^p words except possibly for one subsequence containing fewer words. Thus, the maximum number of passes that may be required to obtain the monotonic nondecreasing sequence through the flow diagram of Fig. 7-2-1 is p where $2^p \geq N$ and $2^{p-1} < N$. By using more tape units this sort could be performed with fewer merges.

The object in presenting the Bucharest sorting program is to demonstrate some of the types of considerations that one may encounter in the programming of problems which involve the use of magnetic tapes. There are more efficient sorting procedures, particularly for the cases where only a few numbers or numbers containing only a few digits are involved.[1] In general, tapes may be used as an auxiliary memory whenever the main memory is not sufficient to hold an entire problem.

[1] H. B. Demuth, "A Report on Electronic Data Sorting," Stanford Research Institute, Menlo Park, Calif., 1956; D. L. Shell, A High-speed Sorting Procedure, *Commun. Assoc. Computing Machinery*, vol. 2, no. 7, pp. 30–32, 1959; R. M. Frank and R. B. Lazarus, A High-speed Sorting Procedure, *Commun. Assoc. Computing Machinery*, vol. 3, no. 1, pp. 20–22, January, 1960; Ivan Flores, Analysis of Internal Computer Sorting, *J. Assoc. Computing Machinery*, vol. 8, no. 1, pp. 41–80, 1961.

7-3. Use of Magnetic Tapes as Input-Output Equipment

The line dividing magnetic tapes as an auxiliary memory from magnetic tapes as an input-output unit is not easily drawn. For example, in the sorting procedure described in the preceding section, if punched cards had been used instead of magnetic tapes, the procedure would have been to punch a deck of cards for each subsequence $\{c_{sk}\}$. For the entering process, the subsequences should be read alternately into the main memory locations for the a_i's and the main memory locations for the b_j's. For a computer with only one card input unit, the actual process will be more complex than just suggested if one should assume that at some stage in the sorting process the subsequences $\{c_{sk}\}$ will exceed the main memory space allotted to either the a_i's or the b_j's. The point, however, is that the sorting process could have been performed with punched cards and input and output units. The unit used to enter information contained in punched cards into a computer is, by custom, called an input unit irrespective of how the information is used in the problem. Similarly, the unit used to transfer information from the computer to punched cards is customarily referred to as an output unit. However, magnetic-tape units are sometimes referred to as an auxiliary memory of a computer and sometimes referred to as input-output units. Determining when a magnetic-tape unit should be referred to as an input-output unit and when it should be referred to as an auxiliary memory is not as important as pointing out the versatility of magnetic-tape units.

Most tape units are constructed so that tapes may be easily removed and loaded. This allows for the transferring of information to a tape, the removing of the tape, and the storing of the tape for future use. Thus, magnetic tapes may be used in business applications where the information stored on the tapes is in the nature of accounts or inventories which must be periodically updated.

Often, permanent information is stored on magnetic tapes. The information contained on such a tape might be a table of the values of a function at known increments of its arguments, or it might be part of a library of subroutines. So that permanent information will not be destroyed by a programming or computer error, which would cause the computer to transfer information to a tape containing permanent information, interlocks are usually placed on tape units. These interlocks may be set so that a tape unit will not obey a write order and so that a write order for a tape unit, the interlock of which has been set, will cause computation to stop.

Since information contained in the main memory may be transferred to magnetic tapes faster than it may be transferred to any other form of

output unit, some computer installations have *off-line* listing equipment (a listing unit which is not an integral part of a computer) which can list the contents of magnetic tapes. By having such equipment, the results of one problem may be listed while another problem is running on the computer. This arrangement allows more computer time for computations than other listing arrangements. Off-line equipment is also used for preparing magnetic-tape input data and programs.

In general, when the information on a tape is variable in nature, i.e., information used by the computer and replaced with new information generated by the computer, the tape unit is considered to be an auxiliary memory. Whereas, when the tape is used to supply input information for a computation or is used to remove the results of a computation from a computer, the tape unit may be considered as an input-output unit. As already mentioned, these definitions are not strong enough to always designate the classification for each use of a tape unit.

PROBLEMS

7-1. Assume either the computer of Appendix I or of Appendix II has four tape units associated with it. Let these units be addressed 001, 002, 003, and 004. Draw a flow diagram which will keep track of the position of the tape in each tape unit and will allow the computer to write or read any specified block on each tape. Assume that blocks may be written over old blocks without rewriting the entire tape. Note that one cannot write information in a block, p blocks from the present position of a tape by writing it on each block without destroying the information previously contained in each of the p blocks.

7-2. Code the flow diagram of Prob. 7-1 for the three-address computer of Appendix I.

7-3. Code the flow diagram of Prob. 7-1 for the two-address computer of Appendix II.

7-4. Considering the computers of Appendices I and II, draw a flow diagram for

$$s_k = \sum_{j=1}^{n_k} a_{kj}$$

assuming floating point operations, tape blocks of 100 words, the a_{kj} existing in floating point representation on the tape of unit 001, $0 \leq n_k \leq 200$, and $1 \leq k \leq 300$. Write the sums s_k in the first three blocks on the tape of tape unit 002.

7-5. Code the flow diagram of Prob. 7-4 for the three-address computer of Appendix I.

7-6. Code the flow diagram of Prob. 7-4 for the two-address computer of Appendix II.

8

SOME ASPECTS OF AUTOMATIC PROGRAMMING

by R. E. Keirstead, Jr.

8-1. Introduction

The term *automatic coding* covers techniques of computer programming which increase the man-hour productivity of a programming and coding staff where productivity is defined as the production of error-free computer routines.

Among such devices are, of course, well-stocked subroutine libraries, diagnostic routines such as memory dumps, snapshot routines, tracing routines, and the like. Also available are various species of interpretive routines in which a program written in a language other than machine code (i.e., the digital codes for the operations of the computer) is continuously translated to machine language, as the program is run, in much the same way that Russian, for example, is continuously translated into English in the United Nations.

We will, however, not dwell on any of these devices. The first mentioned are tools which in the hands of a good programmer provide a bit more coding power. The interpretive routine, while a species of automatic coding, will be omitted since its place is rapidly being taken, at least in the larger computers, by the compiler.

We will discuss two major classes of automatic coding techniques—the assembly routines and the compilers. Both share the common feature that the problem is stated by the coder in a language which is not exactly machine code. The program written by the coder in the new language will henceforth be called the *program*, while the corresponding machine code will be termed the *code*.

8-2. Assembly Routine

The first class, that of the assembly routines, may generally be distinguished from the second class, that of the compilers, in one particular. A thorough knowledge of machine-language coding is needed to write the program. Therefore, routines of the class are often spoken of as automatic coding for the professional. It is common for programs written for processing by routines of this class to be as fast and compact as is possible to code for the machine used.

For most versions of assembly routines, the programmer must write one program step of comparable length and structure for each resulting line of machine code. However, the programmer is permitted a great deal of freedom in the structure of instructions.

Certain techniques are common to all assembly routines. Some of these are the substitution of mnemonic alphabetic order codes for the digital, the avoidance of absolute (i.e., numerical machine) address assignments, and the arrangement for subroutine linkage in a more convenient format.

Absolute addressing can be avoided in several ways. For instructions it is apparent that in the large majority of cases, commands will be executed in the order they are written down. It is also clear that as long as internal consistency is maintained, a set of instructions may validly be executed from any set of memory locations. For these reasons, the computer itself, while it is being used to translate the written program to machine code, makes most absolute address assignments for instructions. Some commands of the program must be referenced by other commands in the same program. For these, and these only, the programmer must supply a location. However, since no numerical location is known a priori, this location may be simply an identifying number, or even better, an alphabetic designator with some meaning in the program, e.g., the first line of the program might aptly be named START, while the first line of a particular part of the program dealing with the calculation of a cube root can be called 3 ROOT, and so on.

For the handling of data and data addresses, several devices are used. For those pieces of data that must, because of machine peculiarities, occupy particular memory positions, the option of assigning an absolute machine memory location must be retained. For the single data word of intermediate computation result, the alphabetic tags permitted for instructions are used. For example, the current value of tan x can be stored in location TAN X. The assembly routine then chooses the absolute memory location that will hold the value of tan x in the running code.

For blocks of ordered data, regional addresses are used. If the set of values d_i is to be entered in the computer in such fashion that the value $d_1, d_2, \ldots, d_N$ occupy N consecutive memory locations, it suffices to name the location of the first of these values. All others in the set are then found in locations displaced relative to the first location. More specifically, the location of d_1 would be $D0001$, while that of d_3 would be $D0003$, etc. In making the final allotment of memory cells for the code, it is sufficient to designate for $D0001$ an absolute address and the others fall into line. The assignment of an absolute value for the address $D0001$ is again left to the assembly routine unless the coder has reasons for doing otherwise. Care must be taken in any case to ensure that sufficient space is allotted by the assembly routine to accommodate all N members of the sequence.

In some assembly routines sets of commands are given regional addresses, enabling one to program in small, somewhat independent sections, each of which is coded in absolute terms relative to the initial command of the segment. It then remains only to assign an absolute location to the initial command of each segment.

To illustrate more fully the use and construction of a typical assembly routine, consider the following example, in which the values

$$s_k = \sum_{i=1}^{10} (\sqrt{c_{ki}} + a_{ki}b_i) \qquad \text{for } k = 1, 2, \ldots, 5$$

are to be calculated. The problem flow diagram is given in Fig. 8-2-1.

For the computer of Appendix I, with card input, card punch, and line printer output, the following coding form is used to write machine coding:

Location	Operation	*A*	*B*	*C*
xxx	xx	xxx	xxx	xxx

Commands written in this format are then punched, one to a card, in any convenient card format. A loading routine is then used to place each command in its associated location.

Written on these sheets, our program, exclusive of details of input-output, is given in Table 8-2-1.

A form for coding with an assembly routine is:

	Location	Operation	*b*	*A*		*B*		*C*	
x	xxx	xxx	x	x	xxx	x	xxx	x	xxx

where each address can have an alphanumeric structure, with the restriction that if the first character of any address is blank or numeric zero, the address is an absolute address. The column b is used to indicate a command requiring B modification. Each operation code is written using the three-letter mnemonic code of Appendix I, e.g., instead of 01,

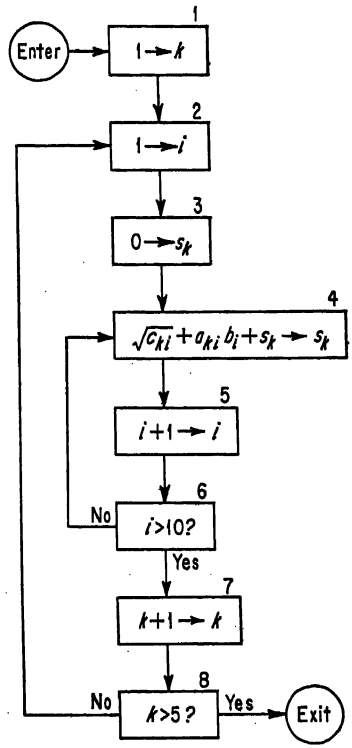


FIG. 8-2-1. Flow diagram for $s_k = \sum_{i=1}^{10} (\sqrt{c_{ki}} + a_{ki}b_i)$, $k = 1, 2, \ldots, 5$.

ADD is used. With this format, the program sheet of Table 8-2-2 would be used in preparing the revised code for key punching.

Having once written the program as shown, it is necessary to change it into an all numeric version acceptable to the computer. This is done by using another computer program—the assembly routine—to make the necessary changes. Some of the changes to be made are obvious; each alphabetic address must be changed to a valid numeric address, each instruction must be given a memory location, and space must be saved

TABLE 8-2-1. Three-address Computer Code for Flow Diagram of Fig. 8-2-1

Location	Operation	A	B	C	Remarks
100	27	000	001	005	$1 \rightarrow k$
101	27	000	002	999	$1 \rightarrow l$
102	27	000	003	010	$1 \rightarrow i$
103	11	000	000	700	$0 \rightarrow s_k$ (loc 700)
104	00	000	000	001	
105	11	000	500	999	$c_l = c_{ki} \rightarrow$ loc 999
106	00	000	002	000	
107	01	000	107	220	Set exit
108	21	000	000	200	SQRT
109	93	550	600	009	$a_{ki}b_i$
110	00	002	003	000	
111	81	999	009	009	$+\sqrt{c_{ki}}$
112	91	009	700	700	$+s_k \rightarrow s_k$
113	00	000	001	001	
114	28	001	002	115	$l + 1 \rightarrow l$
115	28	001	003	105	$i + 1 \rightarrow i$
116	28	001	001	102	$k + 1 \rightarrow k$

Note 1. (000) = 0 always. It may be read out, but no other value can be entered in this memory cell.

Note 2. The subroutine for square root will place the square root of the floating point number in location 999 back into 999 and return control to the location 2 beyond that entered in the B-address position of location 220. Entry to the subroutine is made at location 200.

for the values c_{ki}, a_{ki}, b_i and s_k. From the structure of the coding it is apparent that the values c_{ki} are referred to with the regional addresses $C001$, $C002$, etc., and the values a_{ki}, b_i, and s_k are similarly treated. In order to ensure that enough space is saved to house all the c_{ki}, a_{ki}, b_i and s_k, the only multiple-entry addresses involved, the programmer must provide for each region a *definition card* showing which region is involved, how many entries there are, and where, if it matters, these values are to be placed. These are written as a program step using the dummy operation DEF. For this program the coder must supply four such cards, one each for the regions A, B, C, and S. Let us suppose further that the values of c_{ki} have been calculated at an earlier stage in the computations, while the a_{ki} and b_i are to be read in. In this case one may wish to assign particular memory addresses to the a_{ki} and the b_i, while any unassigned addresses will do for the c_{ki}, s_k. With this in mind we write the four definition cards of Table 8-2-3.

These tell the assembly routine that the values occupying the locations

TABLE 8-2-2. Assembly Program Sheet for the Flow Diagram of Fig. 8-2-1*

Location	Operation	*b*	*A*	*B*	*C*
IN	SET			BX01	005
	SET			BX02	999
LOOP	SET			BX03	010
	ADD				SUM
OVER	ADD	1		*C*001	ARG
				BX02	
TAG	ADD			TAG	EXIT
	UCD				SQRT
	FMR	1	A001	*B*001	TEMP
			BX02	BX03	
	FAD		ANS	TEMP	TEMP
	FAD		SUM	TEMP	SUM
	INC		001	BX02	NEXT
NEXT	INC		001	BX03	OVER
	ADD	1		SUM	S001
					BX01
	INC		001	BX01	LOOP

* This revised program does not follow Fig. 8-2-1 precisely.

TABLE 8-2-3. Definition Cards for a_{ki}, b_i, c_{ki}, and s_k

Location	Operation	*b*	*A*		*B*		*C*	
	DEF		*A*	001		050		950
	DEF		*B*	001		010		940
	DEF		*C*	001		050		
	DEF		*S*	001		005		

addressed *A*001, *A*002, etc., will require 50 locations and must start in machine location 950. Hence locations 950 to 999 are made unavailable for any other use. Similarly the definition card for *B*001 reserves the locations 940 to 949. In the case of the c_{ki} and s_k, we need only identify the regions *C*001 and *S*001, the number of entries, 50 and 5, respectively, and let the assembly routine select the final machine locations wherever convenient.

The commands labeled OVER, et al., are used to enter the square-root subroutine. This routine is written and stored in a subroutine library in the assembly routine format. It is so written that it accepts the floating point value in location ARG and leaves the square root of the value, again in floating point, in location ANS. It is entered at location SQRT. The square-root routine will construct the appropriate exit command if

in the *B* address of location EXIT it finds the location of the command immediately preceding the unconditional transfer to location SQRT. This is accomplished in the coding line labeled TAG in Table 8-2-2. In our particular example there is no need to keep separate the values stored in locations ARG and ANS. To cause the assembly routine to assign the same numeric address to the mnemonic locations ARG and ANS an *equality card* is inserted by the programmer. It, once again, is of the same form as an ordinary command with the pseudo-order code EQU:

Location	Operation	*b*	*A*	*B*	*C*
	EQU		ARG	ANS	

To assemble, the program cards are arranged in the order of running and *preceded* by the definition cards and the equality card. This deck is then read into the computer under control of the assembly routine.

The assembly routine first examines the program cards to establish any definite memory assignments. For this reason the definition cards must be examined first, and, further, the unspecified definition cards must follow those on which a numeric address is specified. With an examination of the definition cards the two parts (*a* and *b*) of Table 8-2-4 can

TABLE 8-2-4. Initial Compiling of Tables for Unavailable Addresses and Equivalent Addresses by Assembly Routine

(*a*) Unavailable addresses	(*b*) Equivalent addresses	
	Mnemonic	Absolute
950–999	*A*001	950
940–949	*B*001	940
890–939	*C*001	890
885–889	*S*001	885

be constructed by the assembly routine. It is assumed here that the assembly routine is so designed that it will make address assignments for data working backward from the end of the memory while for instructions it will work forward from the beginning.

The routine, then, after noting that ARG and ANS can be assigned the same numeric address, proceeds to an examination of the remaining program cards. As it does so it will make the following memory assignments which extend Table 8-2-4 (*b*) above. We assume here that the program step in location IN will be assigned numeric memory address 001.

(As before, location 000 is used with special meaning, i.e., the permanent location of the zero word.)

The new (or extended) Table 8-2-4 (*b*) will, at the conclusion of the first pass, be that of Table 8-2-5. For the assignments starred, note that the

TABLE 8-2-5. Augmented Table of Equivalent Addresses Acquired at End of First Pass

Mnemonic	*Absolute*
*A*001	950
*B*001	940
*C*001	890
*S*001	885
IN	001
BX01	884
BX02	883
LOOP	003
BX03	882
SUM	881
OVER	005
ARG	880
TAG	007
EXIT	879
SQRT	—, 238*
TEMP	878
ANS	880 (=ARG)
NEXT	—, 014*

two addresses, SQRT and NEXT, have been used before an absolute address can be determined for them. Thus a double assignment has been entered in the table. The first encounter with, for example, SQRT causes it to be put in the table, but until it shows up as a member of the square-root code, it cannot be given a numeric equivalent. This implies that following the program listed are library cards containing the program steps for the square-root routine. To be explicit, we assume that at its appearance the line of programming in location SQRT was assigned numeric address 238.

It might be supposed that the address EXIT could not, for similar reasons, be determined until the square-root routine, which uses it, has been encountered. However, the difference is in the use of the addresses. EXIT is the address of a data word as evidenced by the fact that it is the *C* address of an ADD command. Both SQRT and NEXT are addresses of commands, and a free assignment of these cannot be made since they must be considered in a block with other commands that may precede or follow them.

At the end of this look at the program cards, the assembly routine can make the following numerical assignments changing the program to read as shown in Table 8-2-6. For the commands marked with a star we are

TABLE 8-2-6. Computer Code Constructed by the Assembly Routine at the End of First Pass of Program Cards

Location	Operation	*A*	*B*	*C*
001	27	000	884	005
002	27	000	883	999
003	27	000	882	010
004	01	000	000	881
005	11	000	890	880
006	00	000	883	000
007	01	000	007	879
008	21	000	000	SQRT*
009	93	950	940	878
010	00	883	882	000
011	81	880	878	878
012	81	881	878	881
013	28	001	883	NEXT*
014	28	001	882	005
015	11	000	881	885
016	00	000	000	884
017	28	001	884	003

unable to make a final absolute address assignment at the time the starred command is considered. These cases of forward reference must be handled by subjecting this set of intermediate commands to another review in the assembly routine to give any remaining alphabetic addresses a numeric value. At the end of this pass, the program appears as before with the exception of the two starred commands which now have the form:

	008	21	000	000	238
and	013	28	001	883	014

At this time the program is correctly assembled and can be punched out in all numeric form ready for computation.

8-3. The Compilers

Thus far we have talked exclusively of the assembly routine, the chief advantages of which are that for the already-skilled programmer it minimized errors of commission by eliminating many of the details of program

housekeeping. It also provides a more readable version, from the human point of view, of machine code. This minimizes the need for extensive annotation of the machine codes. What is not eliminated, however, is the need to know a great deal about the machine used and about the details of its coding structure.

The second class of automatic programming codes seeks to furnish all the coding aids that the assembly routine allows, and, in addition, to furnish an entirely new language to the programmer. The new language, if skillfully designed, will closely resemble in structure that of the symbolic language of mathematics and at the same time will be single-valued in meaning. The second requirement means that an explicit conversion from the programmer's language to a machine code is possible without ambiguity. Using such a language the sample problem is prepared in the following manner:

```
    REPEAT THROUGH 14: K; 1(1) 5.
    S, K, = 0.
    REPEAT THROUGH 5: I; 1(1) 10.
 5  S, K, = SQRTF (C,KI,) + A, KI, *B, I, + S,K,.
14  DUMMY.
```

This then is a complete description of the program provided certain conventions of writing are understood. The REPEAT THROUGH statement announces an iterative procedure in which each of the statements from the one immediately following to the one named are repeated as the index indicated runs from an initial value to final value in the indicated steps. That is, the statement

$$\text{REPEAT THROUGH } n\text{: } K\text{; } i(\Delta)\ f.$$

means that each of the statements from the next one through that numbered n is repeated on the index K as that index runs from i through f in steps Δ, the period denoting the end of the statement.

The notation $S,K,$ means S_K (that is, the commas set off the subscript). This is needed only because most input media require a linear form of writing. Any alphabetic sequence, e.g., SQRTF, terminating in F, signals a function evaluation, the evaluation being made via a library subroutine, the name of which is the alphabetic sequence, either included by the programmer with the program on punched cards or available in a stored library and brought in by the compiling routine when needed. The argument is defined by the symbols in the parentheses.

The statement designated 5, the main computational statement, is the

program for

$$\sqrt{c_{ki}} + a_{ki}b_i + s_k \rightarrow s_k$$

Comparing these we see that in addition to the subroutine reference and the one-line notation for subscripts, the symbol * is used to denote multiplication rather than either × (which may easily be confused with the alphabetic symbol x) or juxtaposition or the symbol ·. This last symbol could well have been used but is reserved to signal the end of a statement. Hence . means "period" rather than "times."

The statement designated 14 is merely a space filler which is required for operational reasons since one of the restrictions in our compiling routine is that no two REPEAT THROUGH statements may refer to the same terminal statement.

An additional assumption must be made concerning the shape of data. It must be declared, either by the programmer or, in this case, by the compiler itself, how the c_{ki}, for example, are to be stored. Explicitly, we will say that our compiling program will assume (unless otherwise directed) that the values of any singly subscripted symbol, such as u_i are to be stored in locations $U + (i - 1)$, $i = 1, 2, \ldots, I$, using some suitable location U; while doubly subscripted values, as v_{ij}, are found in locations $V + J(i - 1) + (j - 1)$ where $j = 1, 2, \ldots, J$ and $i = 1, 2, \ldots, I$; and so on, for symbols with more subscripts.

The task of the compiling routine is, of course, more complicated than that of the assembly routine. The principal reason for this is, quite naturally, that the program language is considerably further removed from the machine code. The transition, therefore, must be carried out by a program of considerable sophistication.

An analysis of the workings of the compiling routine similar to that of the assembly routine reveals some of the difficulties encountered in extracting from the information in the written program a reasonably efficient machine code. Clearly, several types of examinations of the program will be needed to completely transform the program into a machine code.

For our small example, the first examination will merely give the compiler a chance to examine the memory needs for c_{ki}, a_{ki}, b_i, and s_k. These can be obtained from an examination of the ranges of i and k which are 10 and 5 respectively. With this information the computer can prepare the table of space requirements given in Table 8-3-1. Of course the same result could have been obtained by requiring the programmer to enter, along with the statements comprising the program, definition cards similar to those required in the assembly routine. Indeed, this device elimi-

TABLE 8-3-1. First Space Assignments Made by the Compiler

Item	*Number*
c_{ki}	$50 = K*I$
a_{ki}	50
b_i	10
s_k	5

nates one examination of the program by the compiler. We eliminate the device to show that it is not really needed and that with sufficiently extensive bookkeeping, a good compiler can, in many cases, decipher from statements about subscripts the necessary space requirements of a subscripted variable. The compiler, however sophisticated it may be, cannot make such a decision when the limits of the subscript depend on values computed at the time the compiled program is run.

In any case, some assumption about the way the individual values are stored as functions of the subscripts has to be made by the compiler. Often both methods of memory assignment are allowed, if for no other reason than to permit the programmer to violate the basic assumption about memory layout for a subscripted variable. Whether or not the programmer must always make such an assignment is usually settled by an aesthetic evaluation concerning the relative merits of decreased programmer involvement with a concomitant increase in compilation time, or of a decrease in compiling time achieved by requiring the programmer to supply more details about the data structure of the program.

At the same time the preceding table is established, the compiler can take tentative steps toward expanding the program into machine code. It produces an intermediate program in which some glimmers of the assembly program are apparent (see Table 8-3-2). It is at this point that the assumption about the pattern of storage for a_{ki} and c_{ki} comes into play. The assumption we make is that the value of c_{ki} can be found in location $C + I(k - 1) + (i - 1)$, where I is the maximum value of i. Under this assumption we see that some compounding of the B-boxes containing the current values of k and i is needed to be able to call out the appropriate value of c or a. While it is possible to design a compiling routine to recognize the order in which these values are to be called out in the way a human programmer handled the problem, and then to concoct an extra B-box to handle this, it will in general be easier to prepare the compiler to compute directly the locations of the c_{ki}. That is, given a value k and a value i, to compute the needed increment to the base address to get the address of the desired value. To this end we will need to insert coding for getting the value $I(k - 1) + (i - 1)$ from

TABLE 8-3-2. Intermediate Program

Location	Operation	*b*	*A*	*B*	*C*
	SET		001	BX*K*	005
I.D.	ADD	1	000	000	(*S*001-1)
					BX*K*
	SET		001	BX*I*	010
	*				
5.	ADD	1	000	*C*001	ARG
				BX*KI*	
SELF	ADD		000	SELF	EXIT
	UCD		000	000	SQRT
	FMR	1	(*A*001)	(*B*001-1)	TEMP
			BX*KI*	BX*I*	
	FAD		ANS	TEMP	TEMP
	FAD	1	*S*001	TEMP	*S*001
			BX*K*		BX*K*
	INC		001	BX*I*	5.
	INC		001	BX*K*	I.D.

the values k and i in BX K and BX I respectively. Recalling the structure of the B-box on this computer we find that it is a data word of the form

$$+0\ ccc\ 000\ fff$$

where *ccc* is the current value and *fff* the final value. Our task will be to combine BX K and BX I to create BX KI as follows. If

BX K contains $+0$ $\underline{k}$ 000 $\underline{K}$

while BX I contains $+0$ $\underline{i}$ 000 $\underline{I}$

we will want

BX KI to contain $+0$ $\underline{[I(k-1)+(i-1)]}$ 000 $\underline{xxx}$

where *xxx* are irrelevant digits. The coding to do this, in assembly language, is given in Table 8-3-3. This coding is then inserted at the point starred in the foregoing code along with two extra data words, the extract pattern +0 111 000 000 in location MASK and the value +0 001 000 000 in location ONE. At this point, the entire program is in the language of the assembly routine previously mentioned and from this point on the compiler operates exactly as the assembly routine.

TABLE 8-3-3. Compiler Code Replacing the B-box for the Double Subscript ki

Location	Operation	b	A	B	C
	ADD		000	BXI	TEMP
	XTR		000	MASK	TEMP
	MLT		TEMP	BXK	TMPZ
	SHR		TMPZ + 1	006	TMPZ
	SUB		TMPZ	TEMP	TEMP
	SHL		TEMP	006	TEMP
	ADD		TEMP	BXI	TEMP
	SUB		TEMP	ONE	BXKI

Note that because the compiler routine is unable to analyze the structure of the c_{ki} in the way that a human programmer can, it is necessary to introduce the rather longish patch of coding to produce the needed combination of the values k and i. Thus the code produced by the compiler suffers from verbosity when contrasted to that of the man-made code. This inability on the part of the compiler to perceive a short cut is its principal failing. That they can be made to approximate quite closely the codes of a very skilled programmer is indeed true. But their principal advantage is that they allow one to use a complex computer without spending much time either in learning about the machine or in putting down the myriad details of coding.

What has been noted here about the workings of assembly and compiler routines can merely point out the obvious. The enormous complexities and difficulties inherent in the construction of a good assembly or compiler language and in the details of coding the assembler or compiler can merely be hinted at.

To hypothesize a compiler of any sophistication for a computer of 1,000-word memory, such as that of the computer of Appendix I, is probably academic since the compiler routine needed to diagnose even a simple language such as we presented would undoubtedly occupy thousands of machine locations.

For the small machine, the assembly routine must suffice. Indeed, it is questionable whether the compiler would add a great deal to coding economy. In the case of a large, complex computing machine, the compiler becomes an extremely useful tool, particularly when an open-shop operation is used.

Recently, several new techniques for constructing compilers have been developed, including efficient schemes for deciphering compiler statements. Other techniques, developed for intermeshing the memory lists

required during the compilation, have resulted in efficient combined compiler-assemblers for computers with 4,000-word memories. These compiler-assemblers, which have the combined features of the compiler and the assembler, allow the professional programmer who knows how the computer and compiler work to take full advantage of the machine's capabilities. It is probable that the next generation of digital computers will have an order list designed to include orders to facilitate the design of compiler-assemblers.

8-4. Bibliography

The following lists are by no means complete. However, they may indicate some of the devices that have been used by various programming groups for various machines.

A. Assembly Routines

SOAP II (Symbolic Optimal Assembly Program for the IBM 650; IBM)

GP and GPX (Generalized Programming for the Univac I and II; Remington Rand)

SAC (Semiautomatic Coding for the Datatron 205; Burroughs)

STAR I and II (Assembly programs for the Datatron 220; Burroughs)

SAP (SHARE Assembly Program for the IBM 704; IBM)

These are the routines furnished by the manufacturers for use with their most popular machines. Many other routines exist, both for the above computers, produced by individual users, and for other computers.

B. Compilers

FORTRAN I, II, and III (Formula Translator for the IBM 704 and 709 computers. A restricted version, FORTRANSIT, is available for the IBM 650; IBM)

DATACODE I (A compiler system for the Datatron 205; Burroughs)

MATHMATIC and FLOWMATIC (Compiler programs for the Univac Series of computers; Remington Rand)

IT (The Internal Translator available both for the IBM 650 and the Datatron 205—there referred to as the Purdue Compiler; IBM and Burroughs)

ADES II (Automatic Digital Encoding System II—NAVORD Reports 4209, 4411 and 4412. E. K. Blum, U.S. Naval Ordnance Laboratory, White Oak, Md. The first report describes the encoder logic and could be applied to any machine. The other two reports are concerned with the construction of an IBM 650 program for the encoding)

POGO (Program Optimizer for G-15 Operations. A code for the Bendix G-15 Computer; Bendix)

A more complete list of compilers, interpreters and assemblers can be found in the *Communications of the Association for Computing Machinery* (*CACM*), vol. 1, no. 8, p. 8, August, 1958; vol. 1, no. 11, pp. 5–6, November, 1958; and vol. 2, no. 5, p. 16, May, 1959. Scattered throughout this journal are many other papers and notes on the existence and construction of automatic coding programs.

C. Other

Automatic Coding Monograph No. 3, The *Journal of the Franklin Institute, Philadelphia*, April, 1957; the results of a symposium.

International Algebraic Language (Preliminary Report), A. J. Perlis and K. Samelson, *Commun. Assoc. Computing Machinery*, vol. 1, no. 12, pp. 8-22, Dec. 1, 1958; further discussions of this language are to be found in subsequent issues.

The Arithmetic Translator-Compiler of the IBM FORTRAN Automatic Coding System, P. B. Sheridan, *Commun. Assoc. Computing Machinery*, vol. 2, no. 2, February, 1959. This paper describes rather exactly the process by which FORTRAN translates from its source language to the object program.

E. Grabbe, S. Ramo, and D. Wooldridge (eds.). "Handbook of Automation, Computation and Control," vol. 2, John Wiley & Sons, Inc., New York, 1959. Chapter 2 by J. W. Carr III contains a description of the IT compiler. (This also gives an extensive bibliography and a glossary of computer terminology.)

B. Arden and R. Graham, On GAT and the Construction of Translators, *Commun. Assoc. Computing Machinery*, vol. 2, no. 7, July, 1959.

Association for Computing Machinery Compiler Symposium, *Commun. Assoc. Computing Machinery*, vol. 4, no. 1, January, 1961.

9

A MATHEMATICAL AID FOR PROGRAMMING AND COMPUTER DESIGN

9-1. Introduction

In discussing programming and coding we have emphasized that a systematic approach reduces the number of errors, allows systematic checking of the code and a systematic means of error detection. This systematic approach consists of the translation of the mathematical statement of the problem to a flow diagram and then using the flow diagram for the construction of a computer code. The translation to the flow diagram from the mathematical statement of the problem is by no means a unique procedure, and it is in this operation that judgment exercised by the programmer can make the difference between a good routine and a mere translation of the problem to a computer routine. The construction of a good routine depends to a considerable extent upon the objective of the problem. That is, it may be desirable that the routine not only minimizes the truncation and rounding errors, but also minimizes the memory space required and the computer running time. These four factors, reduction of truncation errors, reduction of rounding errors, reduction of memory space, and reduction of computer running time, frequently compete with one another. For example, the reduction of truncation errors may cause the selection of formulas which increase the number steps in a solution and, thereby, increase the rounding errors, memory-space requirement, and running time of the problem. Similarly, the reduction of memory space for a problem may require choosing computer operations which increase the computer running time and the rounding errors. There are, however, problems for which programming may be selected with the aid of known mathematical procedures. Results for

problems of this type may depend upon, in addition to the usual numerical input data, several logical variables, each of these variables having only two values designated by 1 and 0. For many problems, minimizing the total number of times the decisions (based on the logical variables) are executed in the program will not affect the truncation or rounding errors, but will reduce the computer running time and can also save memory space. Typical of this kind of problem are the payroll calculations where the logical variables determine whether the employee is temporary or permanent, hourly or salaried, has or has not union dues, is or is not a member of the group insurance plan, has or has not retirement deductions, has or has not social security deductions, has or has not state income-tax deductions, has or has not federal income-tax deductions, etc.

In this chapter we introduce a very simple algebra which will satisfy the definition of a Boolean algebra. This same algebra, which is called the *algebra of statements*, may also be used as an aid in the logical design for the construction of a digital computer and in the verification that the computer performs its desired functions. We show first how Boolean and ordinary algebra with an ordered arrangement of the algebraic elements can be used in deriving flow charts.

9-2. Boolean Algebra

The Boolean algebra[1] B consists of the elements $a, b, c, \ldots$, together with two binary operations, the AND operation and the OR operation.[2] The operation or function "a AND b" is written symbolically as

$$a \cdot b$$

and the function "a OR b" is represented by

$$a + b$$

The algebra B also possesses a unary operation NOT[3] and the function "NOT a" is represented by

$$\bar{a}$$

The algebra B also possesses the binary relation of *inclusion* and the

[1] For a further reference see G. Birkhoff and S. MacLane, "A Survey of Modern Algebra," pp. 317–331, The Macmillan Company, New York, 1947.

[2] The AND operation is also referred to as *intersection* or *logical product.* The OR operation is also referred to as the *union* or *logical sum.*

[3] The operation NOT is also called the *complement.*

relation "a included in b" is represented by

$$a \leq b$$

The arbitrary elements $a, b, c, \ldots$, the operations, and relations of the algebra B possess the following properties I through IV.

I. The two binary operations, $\cdot$ and $+$, satisfy the idempotent, commutative, associative, and distributive laws.
   (a) Idempotent laws: $a \cdot a = a$ and $a + a = a$
   (b) Commutative laws: $a \cdot b = b \cdot a$ and $a + b = b + a$
   (c) Associative laws: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
   and $a + (b + c) = (a + b) + c$
   (d) Distributive laws: $a \cdot (b + c) = a \cdot b + a \cdot c$
   and $a + (b \cdot c) = (a + b) \cdot (a + c)$

II. The inclusion relation satisfies the reflexive, antisymmetric, and transitive laws and the consistency principle.
   (e) Reflexive law: for all a, $a \leq a$
   (f) Antisymmetric law: if $a \leq b$ and $b \leq a$, then $a = b$
   (g) Transitive law: if $a \leq b$ and $b \leq c$, then $a \leq c$
   (h) Consistency principle: the three conditions

$$a \leq b$$
$$a \cdot b = a$$
$$a + b = b$$

   are mutually equivalent.

III. There are two elements 0 and 1 which are universal bounds ($0 \leq a \leq 1$, for all a) which satisfy the intersection (the operation "AND") and the union (the operation "OR") laws.
   (i) Intersection laws: $0 \cdot a = 0$ and $1 \cdot a = a$
   (j) Union laws: $0 + a = a$ and $1 + a = 1$

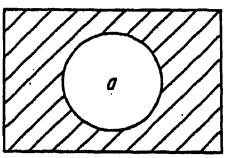
IV. The unary operation NOT obeys the complementary, dualization, and involution laws.
   (k) Complementary laws: $a \cdot \bar{a} = 0$ and $a + \bar{a} = 1$
   (l) Dualization laws: $\overline{a \cdot b} = \bar{a} + \bar{b}$ and $\overline{a + b} = \bar{a} \cdot \bar{b}$
   (m) Involution law: $\bar{\bar{a}} = a$

The properties of this algebra may be represented pictorially by Venn's diagrams[1] as shown in Fig. 9-2-1. In Fig. 9-2-1a, 1 is considered to be the entire interior of the square, a is considered to be the interior of the circle, and $\bar{a}$ is the ruled area (that part of 1 which is not a). An inter-

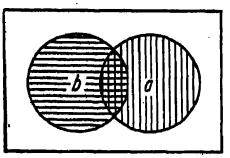
[1] *Ibid.*, pp. 311–314.

pretation of the operations "$\cdot$" and "$+$" is illustrated in Fig. 9-2-1*b*. The region "$a \cdot b$" is only that region which is common to both a and b, whereas the region "$a + b$" consists of all regions contained in either a or b. Thus, $a \cdot b$ is the crosshatched region; $a + b$ consists of the vertically ruled, horizontally ruled, and the crosshatched regions; and it is apparent in this diagram that

$$a \cdot b \leq a + b \qquad a \cdot b \leq a \qquad \text{and} \qquad a \cdot b \leq b$$



(*a*)



(*b*)

FIG. 9-2-1. Venn's diagrams.

Since $0 \leq a$ for all a, 0 represents a null region (region without points).

Figure 9-2-1*a* may be used to illustrate the idempotent, reflexive, intersection, union, complementary, and involution laws; and Fig. 9-2-1*b* may be used to demonstrate the commutative and dualization laws. Similarly, Venn's diagrams may be constructed by the reader to demonstrate the associative, distributive, antisymmetric, and transitive laws and the consistency principle.

9-3. Algebra of Statements

One may verify that there is a Boolean algebra which contains just the two elements 0 and 1. That is, by assuming that 0 and 1 satisfy property III, properties I, II, and IV will also be satisfied. For property III, we have

(*i*) Intersection law (AND operation):

$$0 \cdot a = 0 \Rightarrow 0 \cdot 0 = 0 \quad \text{and} \quad 0 \cdot 1 = 0$$
$$\text{and} \quad 1 \cdot a = a \Rightarrow 1 \cdot 0 = 0 \quad \text{and} \quad 1 \cdot 1 = 1$$

(*j*) Union law (OR operation):

$$0 + a = a \Rightarrow 0 + 0 = 0 \quad \text{and} \quad 0 + 1 = 1$$
$$\text{and} \quad 1 + a = 1 \Rightarrow 1 + 0 = 1 \quad \text{and} \quad 1 + 1 = 1$$

Having the intersection and union laws in terms of the elements 0 and 1, properties I, II, and IV (where $\bar{0} = 1$ and $\bar{1} = 0$) may be verified. For example, the idempotent law is satisfied by both 0 and 1; i.e.,

$$1 \cdot 1 = 1,\ 1 + 1 = 1,\ 0 \cdot 0 = 0, \text{ and } 0 + 0 = 0;$$

and for the eight cases of each of the two distributive laws (in each case

below, the law is underlined), we have

$$0 = 0 \cdot 0 = \underline{0 \cdot (0 + 0) = 0 \cdot 0 + 0 \cdot 0} = 0 + 0 = 0,$$
$$0 = 0 + 0 = \underline{0 + (0 \cdot 0) = (0 + 0) \cdot (0 + 0)} = 0 \cdot 0 = 0;$$
$$0 = 0 \cdot 1 = \underline{0 \cdot (0 + 1) = 0 \cdot 0 + 0 \cdot 1} = 0 + 0 = 0,$$
$$0 = 0 + 0 = \underline{0 + (0 \cdot 1) = (0 + 0) \cdot (0 + 1)} = 0 \cdot 1 = 0;$$
$$0 = 0 \cdot 1 = \underline{0 \cdot (1 + 0) = 0 \cdot 1 + 0 \cdot 0} = 0 + 0 = 0,$$
$$0 = 0 + 0 = \underline{0 + (1 \cdot 0) = (0 + 1) \cdot (0 + 0)} = 1 \cdot 0 = 0;$$
$$0 = 0 \cdot 1 = \underline{0 \cdot (1 + 1) = 0 \cdot 1 + 0 \cdot 1} = 0 + 0 = 0,$$
$$1 = 0 + 1 = \underline{0 + (1 \cdot 1) = (0 + 1) \cdot (0 + 1)} = 1 \cdot 1 = 1;$$
$$0 = 1 \cdot 0 = \underline{1 \cdot (0 + 0) = 1 \cdot 0 + 1 \cdot 0} = 0 + 0 = 0,$$
$$1 = 1 + 0 = \underline{1 + (0 \cdot 0) = (1 + 0) \cdot (1 + 0)} = 1 \cdot 1 = 1;$$
$$1 = 1 \cdot 1 = \underline{1 \cdot (0 + 1) = 1 \cdot 0 + 1 \cdot 1} = 0 + 1 = 1,$$
$$1 = 1 + 0 = \underline{1 + (0 \cdot 1) = (1 + 0) \cdot (1 + 1)} = 1 \cdot 1 = 1;$$
$$1 = 1 \cdot 1 = \underline{1 \cdot (1 + 0) = 1 \cdot 1 + 1 \cdot 0} = 1 + 0 = 1,$$
$$1 = 1 + 0 = \underline{1 + (1 \cdot 0) = (1 + 1) \cdot (1 + 0)} = 1 \cdot 1 = 1;$$
and $$1 = 1 \cdot 1 = \underline{1 \cdot (1 + 1) = 1 \cdot 1 + 1 \cdot 1} = 1 + 1 = 1,$$
$$1 = 1 + 1 = \underline{1 + (1 \cdot 1) = (1 + 1) \cdot (1 + 1)} = 1 \cdot 1 = 1.$$

In a similar manner, the reader may test all of the laws and the consistency principle for the other properties of I, II, and IV.

For programming and for computer logic, it is desirable to consider a set of variables a, b, c, . . . , which are functions of a parameter, say t, and for any specific value of the parameter each variable will have either the value 1 or the value 0. For a payroll problem, t could be the identification number of the employee; and if $a(t)$ is one, employee t is an hourly employee, and if $a(t)$ is zero, employee t is a salaried employee; if $b(t)$ is one, employee t has union dues deducted; and if $b(t)$ is zero, employee t does not have union dues deducted, etc. For computer logic, t could be the running time measured from the beginning of a major computation cycle of the computer to the beginning of the next. If t_s represents the time during a computation cycle at which the sign of the product is determined for a multiplication order and if at this time the computer elements $a(t_s)$, $b(t_s)$, and $c(t_s)$ represent the signs of the multiplier, multiplicand, and product where 1 stands for negative and 0 for positive, then

$$c(t_s) = \overline{a(t_s)} \cdot b(t_s) + a(t_s) \cdot \overline{b(t_s)}$$

is a Boolean equation which determines the sign of the product.

The elements 0 and 1 can be used as a set of elements for a Boolean algebra. Likewise, the set of functions 0, $a(t)$, $b(t)$, $c(t)$, . . . , 1, where the functions can assume only the values 0 or 1, form a set of elements for a Boolean algebra.

9-4. Algebra of Statements as an Aid to Programming

Let us consider a simplified model for the payroll calculations for the monthly pay of the employees of a company. We assume that the possible deductions from the gross pay of each employee are for social security, group insurance, group retirement plan, union dues, and federal income tax. Let the employees be divided into two groups, salaried and hourly. A salaried employee is one who is paid a fixed sum each month, and an hourly employee earns a specified amount each hour and may earn overtime pay according to specified rules. These two groups are further subdivided into types according to company rules and government laws. Let us suppose that the company requires each salaried employee to be a member of the group insurance plan and each salaried male employee, after 1 year of employment if he is thirty years of age or more, to be a member of the group retirement plan. The group retirement plan is optional for salaried female and hourly employees providing they have met the requirements of age and time in employment. The group insurance plan is required for each hourly male employee and is optional for hourly female employees. Let us further assume that the government requires income tax and social security deductions for all employees.

Consider the net monthly earnings, which is the amount of monthly pay, of an employee to be obtained by subtracting the sum of his deductions from his gross monthly earnings. If the employee is salaried, his gross monthly earnings are known. If the employee is hourly, then his hourly wage and the number of hours worked during the month are known and his gross monthly earnings are to be calculated. We will assume that the social security deduction and the federal income tax deduction are calculated from the gross monthly and yearly earnings to date where the gross yearly earnings to date are obtained by adding the gross monthly earnings to the gross yearly earnings to date of the previous pay period. The insurance deduction is obtained from the known amount of insurance carried by the employee and the retirement deduction is determined by the employee's share of the monthly premium for the annuity which is being jointly purchased for the employee by employee and company. Union dues are considered to be a fixed monthly amount.

As a first step in the construction of a flow diagram for the compu-

tation of the monthly pay of each employee, assign a logical variable to each calculation which may have to be performed for an employee; that is, let t be the identification number of the employee, and define

$a(t) = \begin{Bmatrix}0\\1\end{Bmatrix}$ to mean that employee t is $\begin{Bmatrix}\text{salaried}\\\text{hourly}\end{Bmatrix}$ (if hourly, gross monthly earnings must be calculated)

$b(t) = \begin{Bmatrix}0\\1\end{Bmatrix}$ to mean that gross yearly earnings to date of employee t $\begin{Bmatrix}\text{are not}\\\text{are}\end{Bmatrix}$ calculated

$c(t) = \begin{Bmatrix}0\\1\end{Bmatrix}$ to mean that employee t $\begin{Bmatrix}\text{has no}\\\text{has a}\end{Bmatrix}$ retirement deduction

$d(t) = \begin{Bmatrix}0\\1\end{Bmatrix}$ to mean that employee t $\begin{Bmatrix}\text{has no}\\\text{has a}\end{Bmatrix}$ social security deduction

$e(t) = \begin{Bmatrix}0\\1\end{Bmatrix}$ to mean that employee t $\begin{Bmatrix}\text{has no}\\\text{has a}\end{Bmatrix}$ federal income tax deduction

$f(t) = \begin{Bmatrix}0\\1\end{Bmatrix}$ to mean that employee t $\begin{Bmatrix}\text{has no}\\\text{has an}\end{Bmatrix}$ insurance deduction

$g(t) = \begin{Bmatrix}0\\1\end{Bmatrix}$ to mean that employee t $\begin{Bmatrix}\text{has no}\\\text{has}\end{Bmatrix}$ union dues deducted

and

$h(t) = \begin{Bmatrix}0\\1\end{Bmatrix}$ to mean that the net monthly earnings of employee t $\begin{Bmatrix}\text{are not}\\\text{are}\end{Bmatrix}$ calculated

These logical variables have been chosen with a knowledge of the problem. That is, each time a variable is 1, a payroll calculation is required and each time the variable is 0, no calculation is performed. For example, $a(t) = 1$ implies that the employee is hourly and that his monthly gross earnings must be calculated.

With respect to the above logical variables, we can classify each employee according to the type of payroll calculations which must be performed to obtain his net monthly earnings. Suppose that this classification has been done for a company and that the results are as shown in Table 9-4-1.

TABLE 9-4-1. Payroll Classification of Employees

Type	Type	$a(t)$	$b(t)$	$c(t)$	$d(t)$	$e(t)$	$f(t)$	$g(t)$	$h(t)$
1	Salaried (male or female)	0	1	1	1	1	1	0	1
2	Salaried (male or female)	0	1	0	1	1	1	0	1
3	Hourly (male)	1	1	1	1	1	1	1	1
4	Hourly (male or female)	1	1	1	1	1	1	0	1
5	Hourly (male or female)	1	1	0	1	1	1	1	1
6	Hourly (male or female)	1	1	0	1	1	1	0	1
7	Hourly (female)	1	1	0	1	1	0	0	1

By letting t_1 represent any employee's identification number of type 1, t_2 represent any employee's identification number of type 2, etc., the AND operation of the algebra of statements may be used to indicate which payroll calculations are performed and which are not. For the seven types of employees of Table 9-4-1, we have

$$\text{Type 1: } \bar{a}(t) \cdot b(t) \cdot c(t) \cdot d(t) \cdot e(t) \cdot f(t) \cdot \bar{g}(t) \cdot h(t) = \begin{cases} 1 \text{ for } t = t_1 \\ 0 \text{ for } t \neq t_1 \end{cases}$$

$$\text{Type 2: } \bar{a}(t) \cdot b(t) \cdot \bar{c}(t) \cdot d(t) \cdot e(t) \cdot f(t) \cdot \bar{g}(t) \cdot h(t) = \begin{cases} 1 \text{ for } t = t_2 \\ 0 \text{ for } t \neq t_2 \end{cases}$$

$$\text{Type 3: } a(t) \cdot b(t) \cdot c(t) \cdot d(t) \cdot e(t) \cdot f(t) \cdot g(t) \cdot h(t) = \begin{cases} 1 \text{ for } t = t_3 \\ 0 \text{ for } t \neq t_3 \end{cases}$$

$$\text{Type 4: } a(t) \cdot b(t) \cdot c(t) \cdot d(t) \cdot e(t) \cdot f(t) \cdot \bar{g}(t) \cdot h(t) = \begin{cases} 1 \text{ for } t = t_4 \\ 0 \text{ for } t \neq t_4 \end{cases}$$

$$\text{Type 5: } a(t) \cdot b(t) \cdot \bar{c}(t) \cdot d(t) \cdot e(t) \cdot f(t) \cdot g(t) \cdot h(t) = \begin{cases} 1 \text{ for } t = t_5 \\ 0 \text{ for } t \neq t_5 \end{cases}$$

$$\text{Type 6: } a(t) \cdot b(t) \cdot \bar{c}(t) \cdot d(t) \cdot e(t) \cdot f(t) \cdot \bar{g}(t) \cdot h(t) = \begin{cases} 1 \text{ for } t = t_6 \\ 0 \text{ for } t \neq t_6 \end{cases}$$

$$\text{Type 7: } a(t) \cdot b(t) \cdot \bar{c}(t) \cdot d(t) \cdot e(t) \cdot \bar{f}(t) \cdot \bar{g}(t) \cdot h(t) = \begin{cases} 1 \text{ for } t = t_7 \\ 0 \text{ for } t \neq t_7 \end{cases}$$

It may be observed that each of the above algebraic expressions has the value zero when the employee's type is different from that in the left-hand column.

Let us now consider the following OR combination of the preceding seven algebraic expressions:

$$\begin{aligned} &\bar{a}_1 \cdot b_1 \cdot c_1 \cdot d_1 \cdot e_1 \cdot f_1 \cdot \bar{g}_1 \cdot h_1 + \bar{a}_2 \cdot b_2 \cdot \bar{c}_2 \cdot d_2 \cdot e_2 \cdot f_2 \cdot \bar{g}_2 \cdot h_2 \\ &\quad + a_3 \cdot b_3 \cdot c_3 \cdot d_3 \cdot e_3 \cdot f_3 \cdot g_3 \cdot h_3 + a_4 \cdot b_4 \cdot c_4 \cdot d_4 \cdot e_4 \cdot f_4 \cdot \bar{g}_4 \cdot h_4 \\ &\quad + a_5 \cdot b_5 \cdot \bar{c}_5 \cdot d_5 \cdot e_5 \cdot f_5 \cdot g_5 \cdot h_5 + a_6 \cdot b_6 \cdot \bar{c}_6 \cdot d_6 \cdot e_6 \cdot f_6 \cdot \bar{g}_6 \cdot h_6 \\ &\quad + a_7 \cdot b_7 \cdot \bar{c}_7 \cdot d_7 \cdot e_7 \cdot \bar{f}_7 \cdot \bar{g}_7 \cdot h_7 \end{aligned} \quad (9\text{-}4\text{-}1)$$

In the expression (9-4-1), one and only one term will be one for each type of employee, and subscripts have been assigned accordingly. We can now find the main path of calculation by breaking the above expression into its factors using the distributive law. To do this, we determine the frequency of occurrence of each variable in the uncomplemented form. We find that

a occurs 5 times
b occurs 7 times
c occurs 3 times
d occurs 7 times
e occurs 7 times
f occurs 6 times
g occurs 2 times
and h occurs 7 times

Since b, d, e, and h occur in each term uncomplemented, we factor these variables from all terms in (9-4-1) and obtain the expression

$$b \cdot d \cdot e \cdot h \cdot (\bar{a}_1 \cdot c_1 \cdot f_1 \cdot \bar{g}_1 + \bar{a}_2 \cdot \bar{c}_2 \cdot f_2 \cdot \bar{g}_2 + a_3 \cdot c_3 \cdot f_3 \cdot g_3 + a_4 \cdot c_4 \cdot f_4 \cdot \bar{g}_4 + a_5 \cdot \bar{c}_5 \cdot f_5 \cdot g_5 + a_6 \cdot \bar{c}_6 \cdot f_6 \cdot \bar{g}_6 + a_7 \cdot \bar{c}_7 \cdot \bar{f}_7 \cdot \bar{g}_7) \quad (9\text{-}4\text{-}2)$$

where the subscripts have been dropped from the common variables. Within the parentheses of (9-4-2), we again count the number of times each uncomplemented variable occurs and obtain

a occurs 5 times
c occurs 3 times
f occurs 6 times
and g occurs 2 times

This time we factor f from the terms in which it occurs (again the subscript is dropped from common variables) and write (9-4-2) as

$$b \cdot d \cdot e \cdot h \cdot [f \cdot (\bar{a}_1 \cdot c_1 \cdot \bar{g}_1 + \bar{a}_2 \cdot \bar{c}_2 \cdot \bar{g}_2 + a_3 \cdot c_3 \cdot g_3 + a_4 \cdot c_4 \cdot \bar{g}_4 + a_5 \cdot \bar{c}_5 \cdot g_5 + a_6 \cdot \bar{c}_6 \cdot \bar{g}_6) + a_7 \cdot \bar{c}_7 \cdot \bar{f}_7 \cdot \bar{g}_7] \quad (9\text{-}4\text{-}3)$$

Again, within the innermost parentheses of (9-4-3), we count the number of times each variable occurs uncomplemented and obtain

a occurs 4 times
c occurs 3 times
and g occurs 2 times

Factoring a from the terms in which it occurs within the innermost

parentheses of (9-4-3), one obtains

$$b \cdot d \cdot e \cdot h \cdot \{f \cdot [a \cdot (c_3 \cdot g_3 + c_4 \cdot \bar{g}_4 + \bar{c}_5 \cdot g_5 + \bar{c}_6 \cdot \bar{g}_6) + \bar{a}_1 \cdot c_1 \cdot \bar{g}_1 + \bar{a}_2 \cdot \bar{c}_2 \cdot \bar{g}_2] + a_7 \cdot \bar{c}_7 \cdot \bar{f}_7 \cdot \bar{g}_7\} \quad (9\text{-}4\text{-}4)$$

In the innermost parentheses of (9-4-4), both c and g occur twice uncomplemented, and we can again factor with respect to one or the other. If we factor with respect to the variable c, we obtain

$$b \cdot d \cdot e \cdot h \cdot \{|f \cdot \{a \cdot [c \cdot (g_3 + \bar{g}_4) + \bar{c}_5 \cdot g_5 + \bar{c}_6 \cdot \bar{g}_6] + \bar{a}_1 \cdot c_1 \cdot \bar{g}_1 + \bar{a}_2 \cdot \bar{c}_2 \cdot \bar{g}_2\} + a_7 \cdot \bar{c}_7 \cdot \bar{f}_7 \cdot \bar{g}_7|\} \quad (9\text{-}4\text{-}5)$$

The type of employee requiring the greatest number of payroll calculations in order to determine his net monthly earnings is determined by choosing the uncomplemented variable in the innermost bracket. In (9-4-5), g_3 is that variable, and for type 3 we make the calculations associated with the logical variables g, c, a, f, b, d, e, and h. We choose this order as our basic flow path for the payroll calculations as indicated in Fig. 9-4-1. Then, we take the next variable within the same bracket. In this case we have $\bar{g}_4$ and for type 4, we make all calculations except those associated with g. Thus, the calculations for type 4 start at c as indicated in Fig. 9-4-1. After completing the parts of the flow diagram associated with the innermost brackets, we work with the terms in the next innermost brackets, etc. For type 5, we perform those calculations associated with g but not those associated with c. Thus, type 5 enters at box g but skips box c as indicated in Fig. 9-4-1. Type 6 enters at box a since for this type neither the calculations associated with g nor those associated with c are performed. In a similar way the logical flow diagram is completed where all calculations associated with each logical variable are performed from the point of entry to the exit, except when a specifically designated path occurs indicating a variation in calculations to be performed. Each special path carries the number designating the employee type for which the calculations are omitted by that path. The

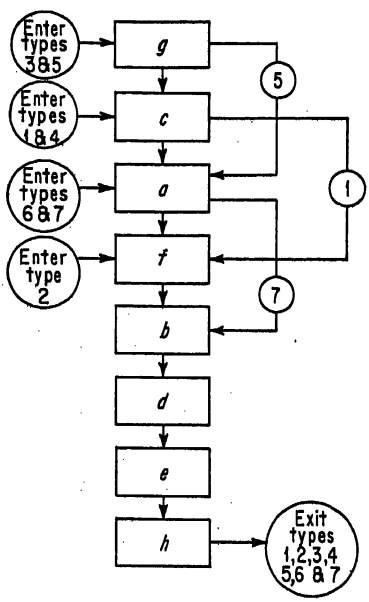


FIG. 9-4-1. Logical flow diagram for independent subroutines in a payroll calculation.

construction of the logical flow diagram of Fig. 9-4-1 assumes that the calculations associated with a particular logical variable are independent of those associated with any other logical variable. It should be noted that the same quantities are to be calculated for each employee type if the flow diagram is reversed—i.e., "exit" is replaced by "enter" and vice versa—and if the directions of the arrows are reversed.

Since the various payroll calculations for the net earnings of each type of employee are not independent of each other, we see that the logical flow diagram of Fig. 9-4-1 will not accomplish the job. That is, the calculations associated with c, d, and e are dependent upon those associated with b, i.e., gross yearly earnings; if the calculations associated with a are to be performed, they must precede those associated with b; and, finally, the calculations associated with h must be the last to be performed. In Fig. 9-4-1, box c comes before boxes a and b and, if the flow diagram is reversed, then boxes d, e, and h come before b which, in turn, comes before a. To obtain the boxes in proper order we return to the original algebraic statements of the calculations which must be performed for each type of employee.

If the ordering of the calculations is important, the problem is divided into parts where the ordering within each part is irrelevant, and each part is treated as a separate problem. Then the parts are tied together in proper order. For the monthly pay of employees, the problem has four parts. The calculations associated with the logical variable $a(t)$ (when they occur) must precede those associated with $b(t)$, and $a(t)$ is considered as the first part and $b(t)$ as the second part. The calculations associated with $c(t)$, $d(t)$, and $e(t)$ must follow those associated with $b(t)$, and $c(t)$, $d(t)$ and $e(t)$ form the third part. Finally, the calculations associated with $h(t)$ must be last and $h(t)$ forms the fourth part. Since the only requirements on the order of the calculations associated with $f(t)$ and $g(t)$ are that they occur before those associated with $h(t)$, these variables may be included in any of the first three parts. Thus, the ordered algebraic expressions for the parts of the payroll calculations could be grouped in the following ways

$$
\begin{gathered}
a,\ b,\ c \cdot d \cdot e \cdot f \cdot g,\ h \\
f \cdot g \cdot a,\ b,\ c \cdot d \cdot e,\ h \\
a,\ f \cdot g \cdot b,\ c \cdot d \cdot e,\ h \\
f \cdot a,\ g \cdot b,\ c \cdot d \cdot e,\ h \\
g \cdot a,\ f \cdot b,\ c \cdot d \cdot e,\ h \\
g \cdot a,\ b,\ c \cdot d \cdot e \cdot f,\ h \\
a,\ g \cdot b,\ c \cdot d \cdot e \cdot f,\ h \\
f \cdot a,\ b,\ c \cdot d \cdot e \cdot g,\ h \\
\text{and} \qquad a,\ f \cdot b,\ c \cdot d \cdot e \cdot g,\ h
\end{gathered}
\tag{9-4-6}
$$

Consider the first grouping of the algebraic statements in (9-4-6) but change it to the following three parts

$$a \cdot b,\ c \cdot d \cdot e \cdot f \cdot g,\ h \tag{9-4-7}$$

The use of the AND operation between a and b instead of treating them as separate groups is done since only two logical variables are involved, and, if the order is important, the logical flow diagram for these variables can be reversed if necessary to obtain the correct order. Writing algebraic expressions of the form of (9-4-7) for each type of employee, one may generate the following algebraic expressions for the three parts of the problem

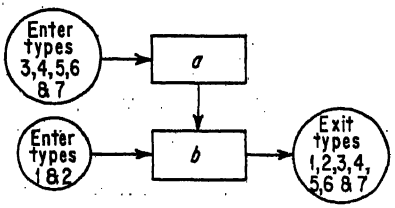


FIG. 9-4-2. Logical flow diagram for first term of Eq. (9-4-9).

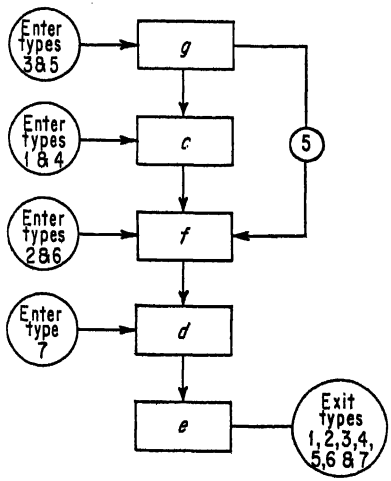


FIG. 9-4-3. Logical flow diagram for second term of Eq. (9-4-9).

$$\begin{aligned} &\bar{a}_1 \cdot b_1 + \bar{a}_2 \cdot b_2 + a_3 \cdot b_3 + a_4 \cdot b_4 + a_5 \cdot b_5 + a_6 \cdot b_6 + a_7 \cdot b_7, \\ &c_1 \cdot d_1 \cdot e_1 \cdot f_1 \cdot \bar{g}_1 + \bar{c}_2 \cdot d_2 \cdot e_2 \cdot f_2 \cdot \bar{g}_2 + c_3 \cdot d_3 \cdot e_3 \cdot f_3 \cdot g_3 \\ &\qquad + c_4 \cdot d_4 \cdot e_4 \cdot f_4 \cdot \bar{g}_4 + \bar{c}_5 \cdot d_5 \cdot e_5 \cdot f_5 \cdot g_5 \\ &\qquad\qquad + \bar{c}_6 \cdot d_6 \cdot e_6 \cdot f_6 \cdot \bar{g}_6 + \bar{c}_7 \cdot d_7 \cdot e_7 \cdot \bar{f}_7 \cdot \bar{g}_7, \\ \text{and}\quad &h_1 + h_2 + h_3 + h_4 + h_5 + h_6 + h_7 \end{aligned} \tag{9-4-8}$$

Factoring each of the expressions of (9-4-8) one obtains

$$\begin{aligned} &(\bar{a}_1 + \bar{a}_2 + a_3 + a_4 + a_5 + a_6 + a_7) \cdot b, \\ &d \cdot e \cdot \{f \cdot [c \cdot (\bar{g}_1 + g_3 + \bar{g}_4) + \bar{c}_2 \cdot \bar{g}_2 + \bar{c}_5 \cdot g_5 + \bar{c}_6 \cdot \bar{g}_6] + \bar{c}_7 \cdot \bar{f}_7 \cdot \bar{g}_7\}, \\ \text{and}\quad &h \end{aligned} \tag{9-4-9}$$

The logical flow diagram for the term $(\bar{a}_1 + \bar{a}_2 + a_3 + a_4 + a_5 + a_6 + a_7) \cdot b$ of (9-4-9), since box a must precede box b, is given in Fig. 9-4-2; and the logical flow diagram for the term

$$d \cdot e \cdot \{f \cdot [c \cdot (\bar{g}_1 + g_3 + \bar{g}_4) + \bar{c}_2 \cdot \bar{g}_2 + \bar{c}_5 \cdot g_5 + \bar{c}_6 \cdot \bar{g}_6] + \bar{c}_7 \cdot \bar{f}_7 \cdot \bar{g}_7\}$$

is given in Fig. 9-4-3. If the partial sum of the deductions is computed as part of each computation associated with each logical variable, then the computation associated with box h is just the subtraction of the total of the deductions from the gross pay. The complete logical flow diagram for the expressions of (9-4-9) is given in Fig. 9-4-4. To complete the analysis, one would construct similar logical flow diagrams for each of the nine ordered algebraic expressions of (9-4-6) and then choose that flow diagram with the least number of special paths.

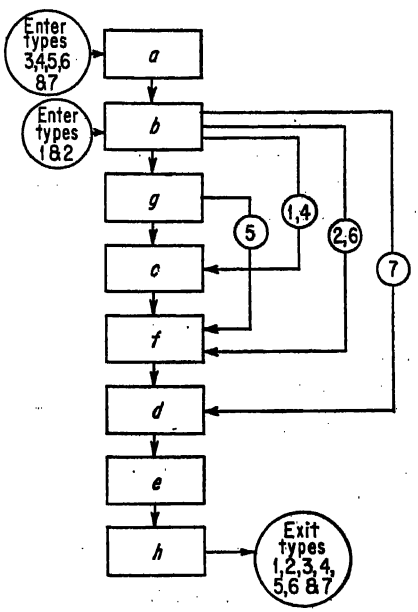


FIG. 9-4-4. A logical flow diagram for the subroutines of a payroll calculation in which partial ordering of the subroutines is required.

In the coding of the detailed flow diagram associated with the logical flow diagram of Fig. 9-4-4, a common procedure for determining if the calculations associated with a given logical variable are to be done or not, for a particular employee, is to test that logical variable to determine if it is or is not one. If the logical variable is one, the calculations are done, and if it is zero, those calculations are skipped and the next logical variable is checked, etc. For example, if t_1 is the identification number of an employee of type 1, then $a(t_1) = 0$ and the calculations associated with $a(t)$ are skipped. Since $b(t_1) = 1$, the calculations associated with $b(t)$ are performed. Thus, a binary decision is made on each logical variable to determine if the calculations associated with that variable are to be performed or skipped.

With respect to the flow diagram of Fig. 9-4-4, we would like to demonstrate how a multiple decision might be used in place of several binary decisions.

9-5. Multiple Branching

Assume $a, b, \ldots, h$ represent subroutines which perform the calculations associated with the logical variables $a(t), b(t), \ldots, h(t)$ respec-

tively. Further, we see from Fig. 9-4-4 that there must be variable exits from subroutines b and g and that the exit of a always leads to the entry of b, the exit of c always leads to the entry of f, the exit of f always leads to the entry of d, the exit of d always leads to the entry of e, and the exit of e always leads to the entry of h. Let $s_1, s_2, \ldots, s_7$ be seven short subroutines which determine those subroutines of a through h which are to be used in the calculations for the net monthly earnings of employee types 1 through 7 respectively. That is, s_1 sets the variable exit of b to the entry of c and causes the computer control to transfer to the first order in the subroutine b; s_2 sets the variable exit of b to the entry of f and causes the computer control to transfer to the first order in the subroutine b; . . . ; and s_7 sets the variable exit of b to the entry of d and causes the computer control to transfer to the first order in subroutine a.

TABLE 9-5-1. A Three-address Code for Multiple Branching

Order symbol I	Order code I	M_1	M_2	M_3	Branch	Memory location	Remarks
	+0	000	000	000		000	
	{+0	000	$0d_5d_6$	$d_7d_8d_9$}		001	Employee identification word containing t
	00	000	000	001		002	Extract constant
	21	000	000	007		003	Constant for UCD command
	{00	000	000	000}		004	Temporary storage of $t = d_9$
XTR	24	001	002	004	Enter	005	XTR $t = d_9$ into (004)
ADD	01	003	004	007		006	Construct UCD command for 007
UCD	{00	000	000	000}	C_{7+t}	007	UCD to 007 + t
UCD	21	000	000	S_1		008	UCD to S_i, the address of the first order in the subroutine s_i, $1 \leq i \leq 7$
UCD	21	000	000	S_2		009	
....	....	...	....	.....		...	
UCD	21	000	000	S_7		014	

Further, let there be an identification word associated with each employee, subdivided so that the digits d_5, d_6, d_7, and d_8 are the identification number of the employee and the digit d_9 is the type number t of the employee. The type numbers of the employees are assumed to be 1, 2, . . . , 7. Let the identification word be stored in memory location 001 of the three-address computer of Appendix I, then the commands of Table 9-5-1 perform a multidecision which selects the proper subroutines for each

employee. The first order performed is in memory location 005. The code in Table 9-5-1 has the virtue that, by minor changes, it may be located anywhere in the memory. If, however, it can be located as indicated in Table 9-5-2, that is, if the type number is the last digit of the memory location of the UCD command for that type number, then the code can be shortened by three words as shown in Table 9-5-2.

TABLE 9-5-2. Alternate Coding for Table 9-5-1

Order symbol I	Order code				Branch	Memory location	Remarks
	I	M_1	M^2	M_3			
UCD	21	000	000	S_1		861	UCD to S_i, the address of the first order in the subroutine S_i, $1 \leq i \leq 7$
UCD	21	000	000	S_2		862	
....	....	...	....	.....		...	
UCD	21	000	000	S_7		867	
XTR	24	870	871	869	Enter	868	XTR $t = d_9$ into (869)
UCD	21	000	000	86{0}	C_{860+t}	869	UCD to $860 + t$
	{+0	000	$0d_5d_6$	$d_7d_8d_9$}		870	Employee identification no. and $t = d_9$
	00	000	000	001		871	Extract constant

9-6. Computer Components and the Algebra of Statements

Computer maintenance routines, in general, test most computer components in some mode of operation under both normal and marginal conditions, i.e., conditions such as low or high operating voltage that increase the likelihood of incipient errors being detected. After such routines are run satisfactorily, the probability that the computer will run until the next scheduled maintenance check, 12 hours to 1 month hence depending upon the computer, is fairly high. However, the maintenance check routines cannot be considered as complete tests for reliable performance of a computer. Considering the number of components in a computer, it would take a prohibitively long time to check all components for all possible sequences of operations. Similarly, even though marginal tests serve the purpose of discovering which computer components are likely to fail first, it is impossible to check before breakdown for all symptoms of failing computer components. Finally, computer components may have transient failures which happen so infrequently that they may not occur during a maintenance check and may occur only intermittently during the run of a long problem. Thus, the programmer must not only find errors in his own programming, but also be able, on occasions, to

prove that his program is correct and that the computer is malfunctioning. (That is, desired or not by the programmer, part of his task is that of determining whether or not the computer is operating properly for the problem he has coded.)

Sometimes the malfunctions of a computer are difficult to observe and are undetected by the check circuits within the computer. Suppose the malfunction is the failure to round a product when a rounded multiplication has been programmed. If the multiplier and multiplicand are near 1 in magnitude, and if the rounding procedure is that of adding 5×10^{-11} to the unrounded product and then retaining that part of the product which, in magnitude, is greater than or equal to 10^{-10}, the computer product will be nearly correct, and a casual observation will not detect the error.

When the programmer has detected a machine error, the speed with which the computer is repaired usually depends upon the completeness of the description of the error. For example, if the computer product of .873256 times .213279 is .186003528000, then, quite obviously, the computer is not performing the multiplication correctly since the last digit of the computer product is 0 instead of 4. However, by observing that the last three digits of the product are 0, it is quite probable that the computer product was obtained either by multiplying .873000 by .213279 or by multiplying .873256 by .213000. Actually, the computer product was equivalent to multiplying .873256 by .213000, which would indicate that the computer may no longer be able to process the last three digits of the multiplicand. Such a condition might be verified by the programmer by varying the last three digits of the multiplicand and checking to see if the computer always produces the same answer when the last three digits are zero as when they are not. Having verified this observation, the programmer can inform the maintenance engineer not only that the computer will not multiply correctly, but also what it appears to do in place of this operation. It is also possible to use general-purpose error-detection routines which test all of the computer commands for a variety of numbers to locate computer failure. If these routines are adequately designed, they can usually isolate errors faster than a human operator.

The more complete the knowledge the programmer possesses of the computer, the more complete his analysis of a machine error can be, and usually the shorter is the time the computer is inoperative and the sooner the programmer has his problem on the computer again. With this in mind, we introduce the relation between the algebra of statements and computer components. For there to be a one-to-one relation between the algebra of statements and the logical design of a computer, one must

be able to recognize, in the computer, electrical circuits which are equivalent to the two binary operations AND and OR, to the unary operation NOT, and be able to recognize the two electronic states which are equivalent to the two algebraic elements 0 and 1.

First, let us consider block diagrams for computer components with only a brief description of how these components may be constructed electronically and then consider how the components are tied together logically to perform specific operations within a computer. Let the AND, OR, and NOT components (gates) be represented pictorially as in Fig. 9-6-1*a*, *b* and *c*, respectively. In Fig. 9-6-1, arrows pointing into a box

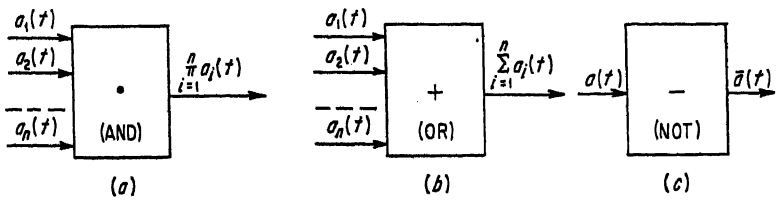


FIG. 9-6-1. AND, OR, and NOT gates.

represent the inputs to the computer component represented by the box and the arrow pointing away from the box is the output of the computer component. Also, let

$$\prod_{i=1}^{n} a_i(t) = a_1(t) \cdot a_2(t) \cdot \cdots a_n(t)$$

and

$$\sum_{i=1}^{n} a_i(t) = a_1(t) + a_2(t) + \cdots + a_n(t)$$

where the center point $\cdot$ and the plus sign $+$ represent the AND and OR operations respectively. At the time $t = \tau$, the output of the logical component of Fig. 9-6-1*a* is

$$\prod_{i=1}^{n} a_i(\tau) = \begin{cases} 1 & \text{if } a_i(\tau) = 1 \text{ for all } i \quad 1 \leq i \leq n \\ 0 & \text{if one or more } a_i(\tau) = 0 \quad 1 \leq i \leq n \end{cases}$$

the output of the logical component of Fig. 9-6-1*b* is

$$\sum_{i=1}^{n} a_i(\tau) = \begin{cases} 1 & \text{if one or more } a_i(\tau) = 1 \quad 1 \leq i \leq n \\ 0 & \text{if } a_i(\tau) = 0 \text{ for all } i \quad 1 \leq i \leq n \end{cases}$$

and the output of the logical component of Fig. 9-6-1*c* is

$$\bar{a}(\tau) = \begin{cases} 1 & \text{if } a(\tau) = 0 \\ 0 & \text{if } a(\tau) = 1 \end{cases}$$

Before discussing an example of a set of typical electronic elements involved in each of the illustrated computer components of Fig. 9-6-1, let us briefly review some of the properties of some electrical circuits. If there is a constant potential difference ΔE between the two ends of a single conductor, then a current i flows along the conductor, the direction of which is said to be from the end at high potential to the end at low potential, and the current is directly proportional to the potential difference ΔE. The constant of proportionality R is known as the resistance of the conductor and is observed to be a physical property of the conductor depending principally upon the physical dimensions of the conductor and the material of which it is made. The preceding statements may be represented by the equation

$$\Delta E = iR \tag{9-6-1}$$

which is known as Ohm's law. The current i is related to the flow of free electrons along the conductor, but the positive direction of i, by convention, is opposite to that of the flow of these electrons. The magnitude of i is assumed to be the same at all points along a single conductor or path. Diagrammatically, a single path of a circuit is shown in Fig. 9-6-2 where

$$\Delta E = E_1 - E_2 \qquad E_1 > E_2$$

and the total resistance is indicated by the saw-tooth symbol at a single position along the symbol for the conductor. In electrical circuits the conductors interconnecting computer components are usually made of materials and are of such dimensions that they have very small resistances (usually so small that this resistance can be neglected) and an electrical element known as a resistor, a conductor of high resistance, may be inserted in the path to regulate the current flowing along the path.

When two or more connectors are jointed at a common point, the connection point is referred to as a *junction*. Kirchhoff's first law states that the algebraic sum of the currents coming to any junction, at which no external potential is applied, in a network of conductors is always zero. That is, at a junction of n conductors,

$$\sum_{k=1}^{n} i_k = 0 \tag{9-6-2}$$

where the positive direction for the current i_k is toward the junction.

Considering the simple network of Fig. 9-6-3, where the potentials at the ends of the network are E_1 and E_2, $E_1 > E_2$, and letting the potential at the junction be designated as E, then by Eq. (9-6-1) we have

$$E_1 - E = i_1 R_1 \quad \text{or} \quad E = E_1 - i_1 R_1 \tag{9-6-3}$$

and

$$E_2 - E = i_2 R_2 \quad \text{or} \quad E = E_2 - i_2 R_2$$

Substituting the second equation of (9-6-3) in the first, we obtain

$$E_1 - i_1 R_1 = E_2 - i_2 R_2 \quad \text{or} \quad E_1 - E_2 = i_1 R_1 - i_2 R_2 \tag{9-6-4}$$

Since the circuit of Fig. 9-6-3 can be considered as a single conductor

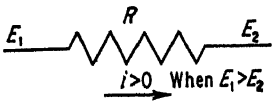


FIG. 9-6-2. Diagram of a single conductor.

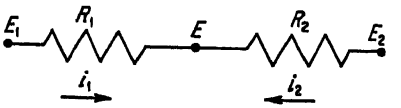


FIG. 9-6-3. A simple series network.

with current i flowing through it, where by Eq. (9-6-2)

$$i = i_1 = -i_2$$

and as having a total resistance R, then Eq. (9-6-4) can be written as

$$E_1 - E_2 = iR$$

where $R = R_1 + R_2$

Thus, the total resistance of two resistors connected in series is the sum of the two resistances. This result is easily extensible to n resistors connected in series and is

$$R = \sum_{k=1}^{n} R_k \tag{9-6-5}$$

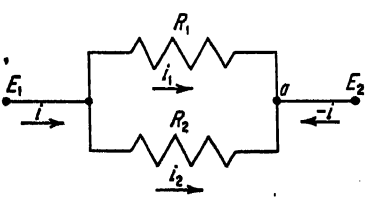


FIG. 9-6-4. A simple parallel network.

Considering the simple network of Fig. 9-6-4, where the potentials at the ends of the network are E_1 and E_2, $E_1 > E_2$, then along each path we have, by Eq. (9-6-1),

$$E_1 - E_2 = i_1 R_1 \quad \text{or} \quad R_1 = \frac{E_1 - E_2}{i_1} \tag{9-6-6}$$

and

$$E_1 - E_2 = i_2 R_2 \quad \text{or} \quad R_2 = \frac{E_1 - E_2}{i_2}$$

Now, let us replace the circuit by an equivalent circuit of the type of Fig. 9-6-2. That is, we wish to replace the two parallel resistors of Fig.

9-6-4 by a single resistor having a resistance R which will cause a current i to flow from E_1 to E_2: we wish to find the value of R such that

$$E_1 - E_2 = iR \qquad \text{or} \qquad \frac{1}{R} = \frac{i}{E_1 - E_2} \tag{9-6-7}$$

For the junction at a in Fig. 9-6-4, by Eq. (9-6-2) we have

$$i = i_1 + i_2$$

And substituting this in Eq. (9-6-7) gives

$$\frac{1}{R} = \frac{i_1}{E_1 - E_2} + \frac{i_2}{E_1 - E_2}$$

or by Eqs. (9-6-6)

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}$$

This result is also extensible to n resistors connected in parallel and is

$$\frac{1}{R} = \sum_{k=1}^{n} \frac{1}{R_k} \tag{9-6-8}$$

In constructing an AND or an OR gate we need to consider another type of electronic element called a *diode*. A diode offers little resistance to the current flow in one direction and a large resistance to current flow in the opposite direction. A diode may be a vacuum-tube diode or a crystal diode. In Fig. 9-6-5, we give the symbol for a crystal diode and show the direction of current flow for small and large resistance where r_d is the small resistance and R_d is the large resistance. These resistances r_d and R_d are called the *forward resistance* and *backward resistance*, respectively.

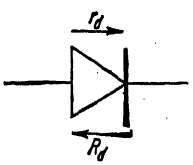


FIG. 9-6-5. Symbol for a crystal diode.

Let us assume that a potential in the range $E_1 + \alpha$ to $E_1 - \alpha$ represents the algebraic element 1 and a potential in the range $E_0 + \beta$ to $E_0 - \beta$ represents the algebraic element 0, where $E_1 - \alpha > E_0 + \beta$. The network of Fig. 9-6-6 is referred to as an AND gate in correspondence with the intersection law of the algebra of statements. The input voltages $A(t)$ and $B(t)$, and the output voltage $C(t)$ represent the potentials corresponding to the values of the algebraic functions $a(t)$, $b(t)$, and $c(t)$ respectively. For example, at time t, if $A(t) \doteq E_1$, i.e., $E_1 - \alpha \leq A(t) \leq E_1 + \alpha$, then $a(t) = 1$ and vice versa, or if $A(t) \doteq E_0$, i.e., $E_0 - \beta \leq$

$A(t) \leq E_0 + \beta$, then $a(t) = 0$ and vice versa. In this circuit it is assumed that the output current requirement i_c is so small that we may assume that $i_c = 0$. Further, we assume that $R_d \gg R \gg r_d$. For the network

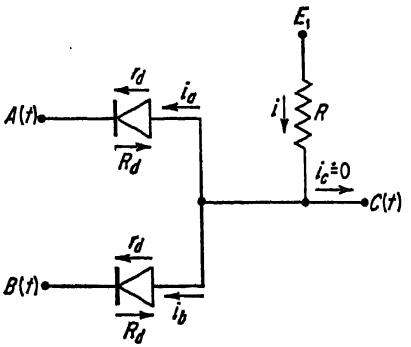


FIG. 9-6-6. An AND gate.

of Fig. 9-6-6, in order to show that $a(t) \cdot b(t) = c(t)$, we show these conditions:

1. If at time t, $A(t) \doteq B(t) \doteq E_1$, then $C(t) \doteq E_1$; that is, if $a(t) = b(t) = 1$, then $c(t) = 1$.
2. If $A(t) \doteq B(t) \doteq E_0$, then $C(t) \doteq E_0$; that is, if $a(t) = b(t) = 0$, then $c(t) = 0$.
3. If $A(t) \doteq E_1$ and $B(t) \doteq E_0$, or if $A(t) \doteq E_0$ and $B(t) \doteq E_1$, then $C(t) \doteq E_0$; that is, if $a(t) = 1$ and $b(t) = 0$ or if $a(t) = 0$ and $b(t) = 1$, then $c(t) = 0$.

For condition 1 the entire circuit is at potential E_1, and there is no flow of current, thus $C(t) = E_1$.

For condition 2, we have

$$i \doteq i_a + i_b, \qquad i_a \doteq i_b \doteq \frac{C(t) - E_0}{r_d}$$

and

$$i \doteq \frac{E_1 - C(t)}{R}$$

Thus

$$\frac{E_1 - C(t)}{R} \doteq \frac{2[C(t) - E_0]}{r_d}$$

or

$$C(t) \doteq \frac{1}{1 + \frac{2R}{r_d}} E_1 + \frac{1}{1 + \frac{r_d}{2R}} E_0 \doteq E_0$$

since $R \gg r_d$.

Since the network is symmetric with respect to $A(t)$ and $B(t)$, we consider only the case $A(t) = E_1$ and $B(t) = E_0$ for condition 3. Here, we have $i \doteq i_a + i_b$

$$i \doteq \frac{E_1 - C(t)}{R} \qquad i_a \doteq \frac{C(t) - E_1}{R_d} \qquad \text{and} \qquad i_b \doteq \frac{C(t) - E_0}{r_d}$$

Thus,
$$\frac{E_1 - C(t)}{R} \doteq \frac{C(t) - E_1}{R_d} + \frac{C(t) - E_0}{r_d}$$

or
$$C(t) \doteq \frac{1}{1 + \frac{R}{r_d}\left(\frac{R_d}{R + R_d}\right)} E_1 + \frac{1}{1 + \frac{r_d}{R} + \frac{r_d}{R_d}} E_0 \doteq E_0$$

since $R_d \gg R \gg r_d$.

The network of Fig. 9-6-6 can be expanded, as in Fig. 9-6-7, to represent the logical box of Fig. 9-6-1*a*.

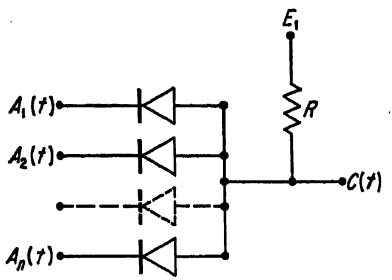


FIG. 9-6-7. Network for Fig. 9-6-1*a*.

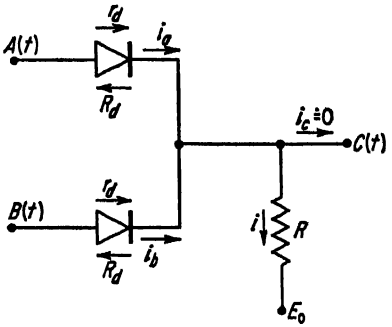


FIG. 9-6-8. An OR gate.

Similarly, the network of Fig. 9-6-8 is referred to as an OR gate in correspondence with the union law of the algebra of statements. For the network of Fig. 9-6-8, in order to show that $a(t) + b(t) = c(t)$, we show the following:

1. If $A(t) \doteq B(t) \doteq E_0$, then $C(t) \doteq E_0$; that is, if $a(t) = b(t) = 0$, then $c(t) = 0$.
2. If $A(t) \doteq B(t) \doteq E_1$, then $C(t) \doteq E_1$; that is, if $a(t) = b(t) = 1$, then $c(t) = 1$.
3. If $A(t) \doteq E_1$ and $B(t) \doteq E_0$ or if $A(t) \doteq E_0$ and $B(t) \doteq E_1$, then $C(t) \doteq E_1$; that is, if $a(t) = 1$ and $b(t) = 0$ or if $a(t) = 0$ and $b(t) = 1$, then $c(t) = 1$.

For condition 1 the entire circuit is at potential E_0 and there is negligible flow of current; thus $C(t) \doteq E_0$.

For condition 2 we have

$$i \doteq i_a + i_b \qquad i_a \doteq i_b \doteq \frac{E_1 - C(t)}{r_d}$$

and

$$i \doteq \frac{C(t) - E_0}{R}$$

Thus,

$$\frac{C(t) - E_0}{R} \doteq \frac{2[E_1 - C(t)]}{r_d}$$

or

$$C(t) \doteq \frac{1}{1 + (2R/r_d)} E_0 + \frac{1}{1 + (r_d/2R)} E_1 \doteq E_1$$

since $R \gg r_d$.

For condition 3, consider as before the case where $A(t) \doteq E_1$ and $B(t) \doteq E_0$; and we have

$$i \doteq i_a + i_b \qquad i_a \doteq \frac{E_1 - C(t)}{r_d} \qquad i_b \doteq \frac{E_0 - C(t)}{R_d}$$

and

$$i \doteq \frac{C(t) - E_0}{R}$$

Thus,

$$\frac{C(t) - E_0}{R} \doteq \frac{E_1 - C(t)}{r_d} + \frac{E_0 - C(t)}{R_d}$$

or

$$C(t) \doteq \frac{1}{1 + \frac{r_d}{R} + \frac{r_d}{R_d}} E_1 + \frac{1}{1 + \frac{R}{r_d}\left(\frac{R_d}{R + R_d}\right)} E_0 \doteq E_1$$

since $R_d \gg R \gg r_d$.

The network of Fig. 9-6-8 can be expanded, as in Fig. 9-6-9, to represent the logical box of Fig. 9-6-1*b*.

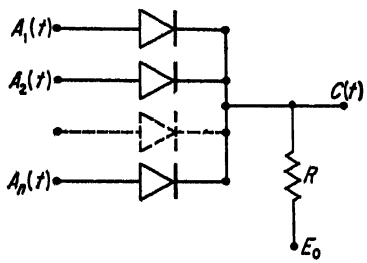


FIG. 9-6-9. Network for Fig. 9-6-1b.

There is a correspondence in the results for the AND gate and the OR gate; *viz.*, if the potential E_0 represents 1 and E_1 represents 0, then Fig. 9-6-6 and Fig. 9-6-7 become OR gates and Figs. 9-6-8 and 9-6-9 become AND gates. This correspondence appears in the formulas derived above.

The description of a NOT gate requires the introduction of another electronic element called the *inverter*. The inverters usually used in computers are either relays, triode vacuum tubes, or transistors. Figure 9-6-10 shows a circuit which is equivalent to the logical box of Fig. 9-6-1c. This circuit is built around a vacuum-tube triode, the functional parts of which are the *plate* or *anode*, the *cathode*, and the *grid*. In normal operation of the tube, the potential E_p of the plate is positive with respect to the potential E_c of the cathode; i.e., $E_p - E_c > 0$; and the grid acts as a valve. When the potential of the grid $G(t)$ is greater than the *cutoff* potential E_g, current flows from plate to the cathode, and the tube is said to be in a *conducting* state. When $G(t) < E_g$, no current flows, and the

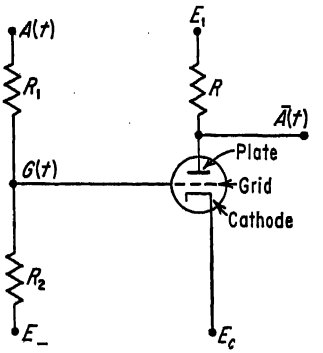


FIG. 9-6-10. A triode used in a network for the NOT gate of Fig. 9-6-1c.

tube is in a *nonconducting* or cutoff state. Depending upon the particular triode tube used, E_g is such that either $E_1 > E_g \geq E_c$ or $E_c > E_g > E_-$. When the tube is conducting, it offers a resistance r to the conducting path from plate to cathode, and the resistance R is chosen such that $\bar{A}(t) \doteq E_0$. When the tube is nonconducting $\bar{A}(t) = E_1$ since no current flows through R. The resistances R_1 and R_2 are chosen such that when $A(t) \doteq E_1$ then $G(t)$ is greater than E_g; and when $A(t) \doteq E_0$, then $G(t)$ is less than E_g. Thus, when $A(t) \doteq E_1$, $\bar{A}(t) \doteq E_0$ (that is when $a(t) = 1$, $\bar{a}(t) = 0$); and when $A(t) \doteq E_0$, $\bar{A}(t) = E_1$ (that is when $a(t) = 0$, $\bar{a}(t) = 1$).

One of the parts of a binary accumulator is called the *logical bit half-adder*. The construction of a half-adder from the three logical computer components of Fig. 9-6-1 is examined next. The half-adder is used to form the sum digit s_k and the carry digit c_k of the two corresponding bits a_k and b_k of the addend and augend respectively. Since a_k, b_k, s_k, and c_k are bits (binary digits), each can assume only the value 0 or 1. We will now determine s_k and c_k as functions of a_k and b_k using the algebra of

statements. First, we construct Table 9-6-1, by the rules of binary addition, which gives s_k and c_k for each of the four possible input combinations of a_k and b_k. From Table 9-6-1 we see that $s_k = 1$ if $a_k = 1$ and $b_k = 0$ or if $a_k = 0$ and $b_k = 1$ and $s_k = 0$ otherwise. Thus,

$$s_k = a_k \cdot \bar{b}_k + \bar{a}_k \cdot b_k$$

The reader can check this algebraic statement against Table 9-6-1.

TABLE 9-6-1. Rules of Binary Addition for a Logical Bit Half-adder

a_k	b_k	s_k	c_k
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

Similarly, $c_k = 1$ if $a_k = b_k = 1$ and $c_k = 0$ otherwise; thus

$$c_k = a_k \cdot b_k$$

Using the logical components of Fig. 9-6-1, a half-adder can be constructed logically with two NOT gates, two AND gates, and an OR gate for s_k and an AND gate for c_k. However, by making use of the identities $\bar{a}_k \cdot a_k = b_k \cdot \bar{b}_k = 0$, we can write s_k as

$$s_k = a_k \cdot \bar{b}_k + b_k \cdot \bar{b}_k + \bar{a}_k \cdot a_k + \bar{a}_k \cdot b_k = (a_k + b_k) \cdot \bar{b}_k + \bar{a}_k \cdot (a_k + b_k)$$
$$= (a_k + b_k) \cdot (\bar{a}_k + \bar{b}_k) = (a_k + b_k) \cdot \overline{(a_k \cdot b_k)}$$

and the half-adder can be constructed, as shown in Fig. 9-6-11, with one

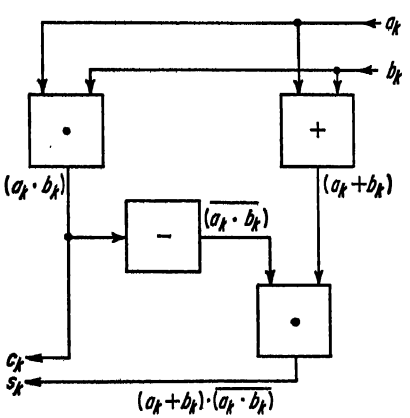


FIG. 9-6-11. Construction of a binary half-adder from AND, OR, and NOT gates.

less gate. A block symbol for the logical circuit of Fig. 9-6-11 is given in Fig. 9-6-12. The half-adder is used in the least significant position of the binary accumulator where no carry bit enters into the sum.

In a similar manner, the three logical computer components of Fig. 9-6-1 can be used to logically design another component, the *logical bit adder*, of a binary accumulator. Suppose we are given two corresponding bits a_k and b_k, of the addend and the augend respectively, and the carry bit c_{k+1}. Here, k represents the bit position within the computer word and $k + 1$ represents the next less significant bit position. The logical adder is required to form the sum bit s_k and the carry bit c_k. Since a_k, b_k, c_{k+1}, s_k, and c_k are bits, each can assume only the value 0 or 1. We wish to write s_k and c_k as algebraic functions of a_k, b_k and c_{k+1} using the algebra of statements. Again, we construct a table, such as Table 9-6-2, from the rules for addition of binary digits, giving s_k and c_k for each of the eight possible input combinations of a_k, b_k, and c_{k+1}.

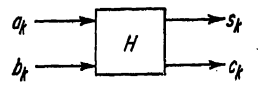


FIG. 9-6-12. Symbol for a binary half-adder.

TABLE 9-6-2. Rules of Binary Addition for a Logical Bit Adder

a_k	b_k	c_{k+1}	s_k	c_k
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

From Table 9-6-2, we see that $s_k = 1$ for four different input combinations and $s_k = 0$ for the other four combinations. The AND combinations of a_k, b_k, and c_{k+1} which cause $s_k = 1$ are

$$a_k \cdot b_k \cdot c_{k+1} \qquad a_k \cdot \bar{b}_k \cdot \bar{c}_{k+1} \qquad \bar{a}_k \cdot b_k \cdot \bar{c}_{k+1} \qquad \text{and} \qquad \bar{a}_k \cdot \bar{b}_k \cdot c_{k+1}$$

The OR combination of these four terms,

$$s_k = a_k \cdot b_k \cdot c_{k+1} + a_k \cdot \bar{b}_k \cdot \bar{c}_{k+1} + \bar{a}_k \cdot b_k \cdot \bar{c}_{k+1} + \bar{a}_k \cdot \bar{b}_k \cdot c_{k+1} \quad (9\text{-}6\text{-}9)$$

causes s_k to be 1 whenever any one of the terms is 1 and causes s_k to be 0 for the other four input combinations. Another algebraic relation which

will cause s_k to be 0, for the proper input combinations, is obtained by writing down the algebraic expression which causes $\bar{s}_k$ to be 1, i.e.,

$$\bar{s}_k = a_k \cdot b_k \cdot \bar{c}_{k+1} + a_k \cdot \bar{b}_k \cdot c_{k+1} + \bar{a}_k \cdot b_k \cdot c_{k+1} + \bar{a}_k \cdot \bar{b}_k \cdot \bar{c}_{k+1} \quad (9\text{-}6\text{-}10)$$

and inverting it to obtain s_k. Similarly, two algebraic expressions which cause c_k and $\bar{c}_k$ to be 1, as required by Table 9-6-2, are

$$c_k = a_k \cdot b_k \cdot c_{k+1} + a_k \cdot b_k \cdot \bar{c}_{k+1} + a_k \cdot \bar{b}_k \cdot c_{k+1} + \bar{a}_k \cdot b_k \cdot c_{k+1} \quad (9\text{-}6\text{-}11)$$

and

$$\bar{c}_k = \bar{a}_k \cdot \bar{b}_k \cdot \bar{c}_{k+1} + \bar{a}_k \cdot \bar{b}_k \cdot c_{k+1} + \bar{a}_k \cdot b_k \cdot \bar{c}_{k+1} + a_k \cdot \bar{b}_k \cdot \bar{c}_{k+1} \quad (9\text{-}6\text{-}12)$$

Using the laws of Boolean algebra, Eqs. (9-6-9) to (9-6-12) may be written in many equivalent forms. For example,

$$\begin{aligned} s_k &= \bar{c}_{k+1} \cdot (\bar{a}_k \cdot b_k + a_k \cdot \bar{b}_k) + c_{k+1} \cdot (a_k \cdot b_k + \bar{a}_k \cdot \bar{b}_k) \\ &= \bar{c}_{k+1}[(\bar{a}_k + \bar{b}_k) \cdot (a_k + b_k)] + c_{k+1}[(a_k \cdot b_k) + \overline{(a_k + b_k)}] \\ &= \{\overline{c_{k+1} + [(a_k \cdot b_k) + \overline{(a_k + b_k)}]}\} + \{c_{k+1} \cdot [(a_k \cdot b_k) + \overline{(a_k + b_k)}]\} \end{aligned} \quad (9\text{-}6\text{-}9a)$$

$$\bar{s}_k = \bar{c}_{k+1} \cdot (a_k \cdot b_k + \bar{a}_k \cdot \bar{b}_k) + c_{k+1}(\bar{a}_k \cdot b_k + a_k \cdot \bar{b}_k) \quad (9\text{-}6\text{-}10a)$$

or

$$\begin{aligned} s_k &= \{\overline{\bar{c}_{k+1} \cdot [(a_k \cdot b_k) + \overline{(\bar{a}_k \cdot \bar{b}_k)}]}\} \\ &\qquad \cdot \{\overline{c_{k+1} \cdot (\bar{a}_k \cdot b_k + \bar{a}_k \cdot a_k + a_k \cdot \bar{b}_k + b_k \cdot \bar{b}_k)}\} \\ &= \{c_{k+1} + [\overline{(a_k \cdot \bar{b}_k)} \cdot (a_k + b_k)]\} \cdot \{c_{k+1} \cdot [\overline{(a_k \cdot \bar{b}_k)} \cdot (a_k + b_k)]\} \end{aligned} \quad (9\text{-}6\text{-}10b)$$

$$\begin{aligned} c_k &= a_k \cdot b_k \cdot (c_{k+1} + \bar{c}_{k+1}) + c_{k+1} \cdot (a_k \cdot \bar{b}_k + \bar{a}_k \cdot b_k) \\ &= a_k \cdot b_k + c_{k+1} \cdot [\overline{(a_k \cdot \bar{b}_k)} \cdot (a_k + b_k)] \end{aligned} \quad (9\text{-}6\text{-}11a)$$

and

$$\begin{aligned} \bar{c}_k &= \bar{a}_k \cdot \bar{b}_k + \bar{c}_{k+1} \cdot [(\bar{a}_k + \bar{b}_k) \cdot (a_k + b_k)] \\ &= \overline{(a_k + b_k)} + \bar{c}_{k+1} \cdot \overline{[(a_k \cdot b_k) + \overline{(a_k + b_k)}]} \end{aligned} \quad (9\text{-}6\text{-}12a)$$

or

$$\begin{aligned} c_k &= \overline{\overline{(a_k + b_k)} + \{c_{k+1} + [(a_k \cdot b_k) + \overline{(a_k + b_k)}]\}} \\ &= (a_k + b_k) \cdot \{c_{k+1} + [(a_k \cdot b_k) + \overline{(a_k + b_k)}]\} \end{aligned} \quad (9\text{-}6\text{-}12b)$$

Our objective is to construct, using the logical components of Fig. 9-6-1, a computer component with the least number of AND, OR, and NOT gates which will form s_k and c_k from the inputs a_k, b_k, and c_{k+1}.

There are many equivalent Boolean expressions for any particular expression and general procedures for determining the expression which produces the minimum number of logical components for a given set of

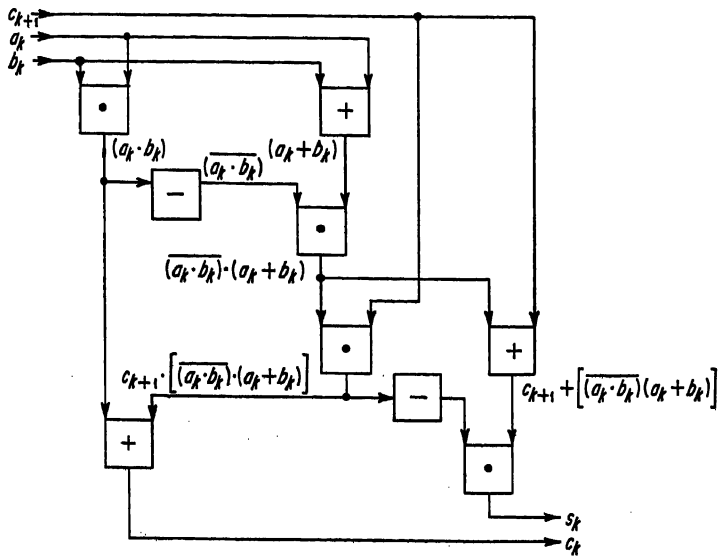


FIG. 9-6-13. Construction of binary adder from AND, OR, and NOT gates.

inputs have not been developed. Thus, the circuit given in Fig. 9-6-13 may not be the best choice, but it is considered a reasonably good one. Engineering aspects, such as the number of levels of gates that have to be switched, are relevant when the speed of operation is considered. Figure 9-6-13 is constructed from Eqs. (9-6-10*b*) and (9-6-11*a*). By comparing Eqs. (9-6-9*a*) and (9-6-12*b*) with Eqs. (9-6-10*b*) and (9-6-11*a*), one can see that another form of the logical binary adder can be constructed with the same total number of gates. A block symbol for the logical binary adder circuit is given in Fig. 9-6-14. The adder is used in all digit positions, except the least significant, of the binary accumulator.

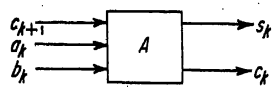


FIG. 9-6-14. Symbol for a binary adder.

Registers in a computer are used as temporary storage for computer words, or digits of a computer word. In the arithmetic unit, registers are used to hold one or both of the operands and the result of the arithmetic operation. In the control unit of the computer there is usually a register which holds the command that the computer is performing and a register containing the address of the next command that the computer is to perform. Registers are often used for *B*-boxes and as buffer storage for input and output equipment for computers. A general requirement

for a register is that its contents are changed to that of a new word being entered, but the use of the information contained within the register does not change its contents. Each bit position of the register is composed of a computer component known as a *flip-flop*. In a computer, the flip-flop is a bistable component which stores not only the bit but also its complement.[1] A simple logical diagram for a flip-flop can be constructed from the computer components of Fig. 9-6-1. Such a logical diagram is given in Fig. 9-6-15 where the input digits, d and e, are transmitted simultaneously to the flip-flop. In the operation of this flip-flop, it is assumed that $d = e = 0$ whenever a digit and its complement are not being read into the flip-flop. If a digit and its complement are not being read into the flip-flop and if the output of OR gate 1 is one, then the two inputs to OR gate 2 are 0 and its output is 0, which is inverted and fed back to

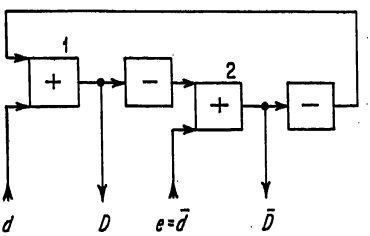


FIG. 9-6-15. Logical construction of a flip-flop.

OR gate 1. Thus, the flip-flop remains in this state; i.e., output D of OR gate 1 is equal to one, and the output of $\bar{D}$ OR gate 2 is equal to zero. Similarly, if $d = e = 0$ and if the output D of OR gate 1 is zero, then the input to OR gate 2 from the inverter is equal to one and the output of OR gate 2 is equal to one, which is fed back through an inverter to OR gate 1 as zero. Thus, the flip-flop remains in this state; i.e., output D of OR gate 1 is equal to zero and output $\bar{D}$ of OR gate 2 is equal to one. However, if the output of OR gate 1 is equal to one when $d = 1$ and $e = \bar{d} = 0$ are gated into the flip-flop, then the output D of OR gate 1 remains one, and the output $\bar{D}$ of OR gate 2 remains zero; but if $d = 0$ and $e = \bar{d} = 1$ are gated into the flip-flop, then the output $\bar{D}$ of OR gate 2 becomes one, and the two inputs to OR gate 1 become zero, and its output D becomes zero. Similarly, if the output D of OR gate 1 is zero then it remains so when $d = 0$ and $e = \bar{d} = 1$ are gated into the flip-flop, but the output D of gate 1 becomes one if $d = 1$ and $e = \bar{d} = 0$ are gated

[1] Some computers have storage elements capable of several discrete states; e.g., the Decatron has ten stable states.

into the flip-flop. The bit stored in the flip-flop may be read out at D and its complement read out at $\bar{D}$. Figure 9-6-16 represents a logical symbol for a flip-flop component where d and $\bar{d}$ are the inputs and D and $\bar{D}$ are the outputs.

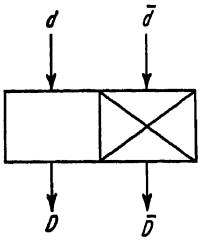


FIG. 9-6-16. Symbol for a flip-flop.

Using AND gates, NOT gates, flip-flops, a half-adder, and adders, we can illustrate in Fig. 9-6-17 the logical construction of that part of a binary accumulator which can perform the sum of the magnitudes of two words. In this figure the sum digits are given by S_1 through S_n, and if an overflow occurs then $C_0 = 1$. The input g is a gating (timing) pulse which is 1 for the appropriate portion of the computation cycle at which time the sum is performed and $g = 0$ at all other times.

Using an OR gate, two AND gates, binary half-adders and binary adders, a simple binary-coded decimal adder may be logically constructed. As demonstrated in the chapter on number representations, ten of the 16

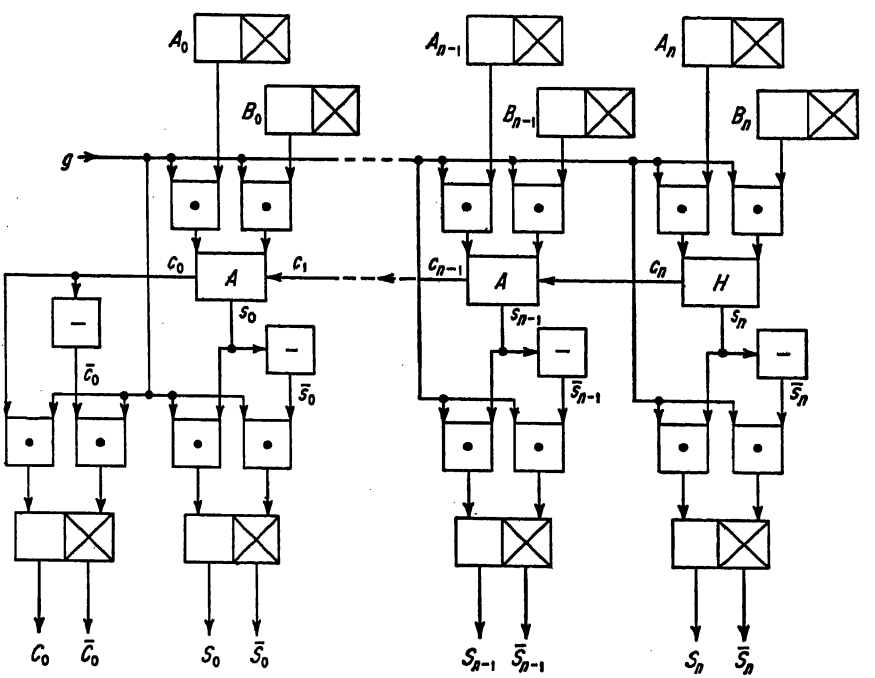


FIG. 9-6-17. Binary word adder.

possible combinations of four bits may be used to represent the ten decimal digits. That is, if A_k is the kth digit of a decimal word then

$$A_k = A_{k3}2^3 + A_{k2}2^2 + A_{k1}2 + A_{k0}$$

where the A_{ki}, $i = 0, 1, 2, 3$, are the bits. Consider the logical design of a binary-coded decimal adder which forms the binary-coded decimal sum digit S_k and the carry bit C_k from an augend digit A_k, an addend digit B_k, and the carry bit C_{k+1}. We may construct a table showing S_k and C_k for all of the possible ways of performing the arithmetic sum $A_k + B_k + C_{k+1}$. Such a table would have the binary-coded representations for A_k, B_k, and S_k. We have already seen in the chapter on number representations that, independent of the number base, C_k and C_{k+1} can only take on the values 0 or 1. Such a table would have 200 entries and from it we would write an algebraic statement for each of S_{k3}, S_{k2}, S_{k1}, and S_{k0}, and C_k. To write each statement we would have to consider as many as 200 possible terms. Rather than perform this rather tedious task, we construct Table 9-6-3

TABLE 9-6-3. Sums for Decimal and Hexadecimal Digit Addition

Decimal						Hexadecimal					
C_k	S_k	S_{k3}	S_{k2}	S_{k1}	S_{k0}	C'_k	S'_k	S'_{k3}	S'_{k2}	S'_{k1}	S'_{k0}
0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	1	0	0	0	1
0	2	0	0	1	0	0	2	0	0	1	0
0	3	0	0	1	1	0	3	0	0	1	1
0	4	0	1	0	0	0	4	0	1	0	0
0	5	0	1	0	1	0	5	0	1	0	1
0	6	0	1	1	0	0	6	0	1	1	0
0	7	0	1	1	1	0	7	0	1	1	1
0	8	1	0	0	0	0	8	1	0	0	0
0	9	1	0	0	1	0	9	1	0	0	1
1	0	0	0	0	0	0	a	1	0	1	0
1	1	0	0	0	1	0	b	1	0	1	1
1	2	0	0	1	0	0	c	1	1	0	0
1	3	0	0	1	1	0	d	1	1	0	1
1	4	0	1	0	0	0	e	1	1	1	0
1	5	0	1	0	1	0	f	1	1	1	1
1	6	0	1	1	0	1	0	0	0	0	0
1	7	0	1	1	1	1	1	0	0	0	1
1	8	1	0	0	0	1	2	0	0	1	0
1	9	1	0	0	1	1	3	0	0	1	1

which contains all of the possible combinations of the decimal sum digit S_k with its binary equivalent S_{k3}, S_{k2}, S_{k1}, S_{k0} and the carry bit C_k showing the equivalent hexadecimal sum digit S'_k, with its binary equivalent S'_{k3}, S'_{k2}, S'_{k1}, S'_{k0}, and carry bit C'_k. From this table we observe that $C_k = C'_k$ and $S_k = S'_k$ for $0 \leq S'_k \leq 9$ and $C'_k = 0$; $C_k = \bar{C}'_k = 1$ and $S_k = S'_k - a$ or equivalently $2^4 + S_k = S'_k + 6$ for $a \leq S'_k \leq f$; and $C_k = C'_k$ and $S_k = S'_k + 6$ for $0 \leq S'_k \leq 3$ and $C'_k = 1$. Also from Table 9-6-3, we see that $C_k = \bar{C}'_k$ when and only when

$$S'_{k3} = 1 \qquad \text{and} \qquad \begin{cases} S'_{k2} = 1 \\ \text{or} \\ S'_{k1} = 1 \end{cases} \tag{9-6-13}$$

otherwise $C_k = C'_k$. Furthermore, we may write in binary arithmetic

$$2^4\bar{C}'_kC_k + S_{k3}S_{k2}S_{k1}S_{k0} = S'_{k3}S'_{k2}S'_{k1}S'_{k0} + 0110C_k \tag{9-6-14}$$

Thus, a binary-coded decimal adder may be constructed by altering a binary-coded hexadecimal adder.

A binary-coded hexadecimal digit adder is a binary accumulator containing four bits [see Eq. (3-9-6)] as shown in the top part of Fig. 9-6-18. We make two alterations to this adder. The first is the formation of C_k by the algebraic statement

$$C_k = C'_k + S'_{k3} \cdot S'_{k2} + S'_{k3} \cdot S'_{k1} = C'_k + S'_{k3} \cdot (S'_{k2} + S'_{k1}) \tag{9-6-15}$$

which is equivalent to the condition of (9-6-13). The other alteration is to use Eq. (9-6-14) when $C_k = 1$ and set $S_k = S'_k$ when $C_k = 0$. The binary-coded decimal adder is given in Fig. 9-6-18.

If we had attempted the design of the binary-coded decimal adder of Fig. 9-6-18 as a problem in the algebra of statements, we might find a computer very helpful. The computer would be programmed to execute the laws and operations of the algebra of statements so that it would create the statements for each of the output variables $S_{k3} = 1$, $\bar{S}_{k3} = 1$, $S_{k2} = 1$, $\bar{S}_{k2} = 1$, $S_{k1} = 1$, $\bar{S}_{k1} = 1$, $S_{k0} = 1$, $\bar{S}_{k0} = 1$, $C_k = 1$, and $\bar{C}_k = 1$ as functions of C_{k+1}, A_{k0}, B_{k0}, A_{k1}, B_{k1}, A_{k2}, B_{k2}, A_{k3}, and B_{k3}. The computer program would then perform a sequence of algebraic operations using the distributive law, the associative law, the commutative law, the idempotent law, etc., repeating the sequence until each expression could not be reduced further by the computer. Since there are no general mathematical techniques for obtaining algebraic statements which represent a minimum number of logical computer components, the results of

the computer might be further analyzed through human effort.[1] It is interesting to note, however, that the logical design of Fig. 9-6-18 was obtained through a knowledge of number representations in bases 2, 10, and 16 and of the algebra of statements. Although Fig. 9-6-18 may not be a minimal design, it is sufficiently compact to be used in the design for a binary-coded decimal adder.

We have demonstrated how one might proceed in the logical design of an arithmetic unit of a computer and, although this knowledge is not essential for one primarily interested in coding and programming, it is

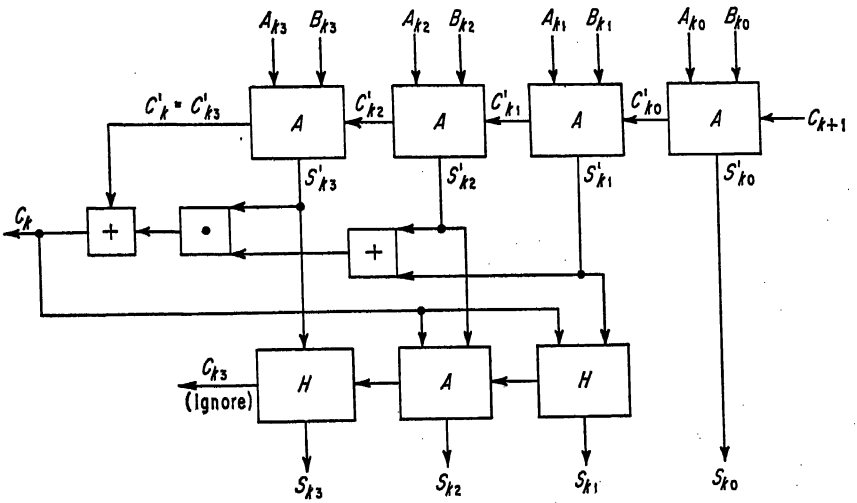


FIG. 9-6-18. Binary-coded decimal digit adder.

knowledge which may be useful to him. If in checking a particular problem the desired answers are not obtained, the error may occasionally be one of the computer and not of the program. For example, in Fig. 9-6-18, if the top AND gate is malfunctioning and its output is always zero, then a wrong answer may be one in which C_k is zero instead of one and S_k a base-16 number greater than or equal to $c = 12_{10}$. For a decimal computer such a configuration is called an *invalid combination* or invalid code. Often, check circuits are added to computers to stop its operation on the occurrence of invalid combinations. In Fig. 9-6-13, if the output of the top left AND gate is always zero, then c_k will be zero instead of one when-

[1] A. Freilich, Eliminating Diode Redundancy in Encoding and Decoding Matrices, *Control Eng.*, pp. 110–114, June, 1960.

ever $a_k = b_k = 1$. Other examples can be constructed by the reader; however, our purpose here is to demonstrate that if the programmer develops a knowledge of the computer, he will often be able to recognize a computer error and save himself the effort of trying to debug a correct program.

Another important reason for the programmer to become acquainted with Boolean algebra is that the automatic compilers currently being developed, such as ALGOL, use Boolean expressions for the control of the computer sequence.

10

NUMERICAL ANALYSIS

10-1. Introduction

The area of mathematics called numerical analysis is concerned with procedures for calculating numerical results for mathematically formulated problems and with the properties of these procedures. The objective of obtaining numerical results distinguishes numerical analysis somewhat from the other branches of mathematics[1] where the objective may be the determination of properties of solutions or the development of general theories.

Numerical analysis is an old field with an extensive literature. Recently, however, there have been marked changes in emphasis and increased interest in numerical analysis resulting from both the widespread use of automatic computers, which greatly increases the amount of computation that is economically feasible, and the increased use of mathematics in technical applications. One such change is evidenced by the study of iterative procedures and other procedures requiring a large amount of computation, e.g., the inversion of large matrices. There is emphasis on the study of general procedures that are applicable to a variety of situations and emphasis on studies of the amount of approximation error introduced by the numerical procedure.

Since a general discussion of numerical analysis is beyond the scope of this book, this chapter discusses mainly the numerical analysis for a few of the examples used in this text. The chapter starts with a discussion of rounding errors that are introduced during a computation. A bibliography of books on numerical analysis that are concerned with automatic computation concludes the chapter.

[1] R. Bellman, in James Glenn (ed.), "The Tree of Mathematics," chap. 26, Digest Press, Pacoima, Calif., 1957.

The analysis of most problems for which digital computation is required falls into four interrelated phases:

1. The formulation and analysis of the mathematical model
2. Formulation of numerical methods
3. Programming and coding of the numerical procedure for the computer
4. Analysis of the results

The completeness of the mathematical model may depend on the existence of analytic and numerical procedures and the speed and memory capacity of the computer. If several numerical procedures are available, the procedure is chosen for accuracy of the results, simplicity of programming, and speed of computation. The computation sequence frequently effects the accumulation of computational errors and thus may become an important part of the numerical analysis. The state of development of numerical analysis thus may influence all phases of a problem from the formulation to the programming details and analysis of the results. It is important that the programmer know the methods that are suitable for his problem and the properties of the methods he uses.

For automatic computations it is generally necessary to have a more complete formulation of a problem than when automatic equipment is not used. For example, in addition to the mathematical process sufficient to determine a unique solution, it may be necessary to have information about the properties of the solution such as upper and lower bounds, rate of oscillation, rate of convergence, monotonicity, or asymptotic behavior. Such information is useful in programming the computations and providing numerical checks for the results.

The formulation of problems and choice of numerical methods for a computer routine should provide for as much generality as is practicable. The reason for this generality is to allow for subsequent computations to produce additional results, or for use of part or all of the computer routine in other problems. In automatic computation, complicated numerical procedures or methods involving human intervention are to be avoided whenever practicable.

10-2. Significant Digits

In the base b, where

$$x^* = \sum_{i=l}^{m} d_i b^i$$

is the digital approximation for a number x, the digit d_i, is called a significant digit or reliable digit if

$$|x - x^*| \leq \tfrac{1}{2}b^i$$

The number of digits in an approximation that are reliable is called the number of significant figures or digits.

The error ϵ of an approximation is variously defined as $x^* - x$, $x - x^*$, or $|x^* - x|$. The definition

$$\epsilon = x - x^*$$

will be used here; that is, the error ϵ in an approximation is the amount to be added to the approximation to obtain the exact value.

The relative error p for an approximation will be defined as

$$p = \frac{x - x^*}{x^*} = \frac{\epsilon}{x^*} \qquad x^* \neq 0$$

The number of significant digits n in an approximation is related to the magnitude of the relative error in that

$$n - 1 \leq \log_b \frac{1}{2|p|} < n + 1$$

10-3. Rounding

In digital computing, storage space and computing time (e.g., multiplication and division times depend on the number of digits in the multiplier and quotient) may be saved by reducing the number of digits used in intermediate results. When successive operations are exact, the number of digits in the result may increase rapidly. For example, raising a 10-digit number to the fifth power produces a result from 46 to 50 digits in length. Division of a 10-digit number by another 10-digit number may produce an unending sequence of digits. Reducing the number of digits generally increases the approximation error and these approximation errors in general propagate throughout the computation. Thus the number of digits retained in a computation is usually a compromise between accuracy and computing capacity. Most electric desk calculators have 10-decimal-digit keyboards and many automatic computers have 10-decimal-digit storage registers. For these computers, retention of more than ten digits in all intermediate results usually far more than doubles the computation time. It has been repeatedly recommended that the digit length be increased with increasing speed and memory capacity of computers so that as the computation complexity increases

there is provision for additional precision to compensate for the additional propagation of error.

There are several procedures for reducing the number of digits in the approximation for a number. In the process called chopping[1] all unwanted digits are merely deleted. In most of the processes called rounding a new approximation x^* for the number x is obtained such that

$$|x - x^*| \leq (\tfrac{1}{2})b^p$$

where b^p is the base raised to the power associated with the last digit kept in the new approximation x^*. In commercially manufactured automatic computers and in the newer electric desk calculators, the round operation (also called half-adjust in some computer manuals) is accomplished by adding $(\tfrac{1}{2})b^p$ to the magnitude of x and then dropping the unwanted digits. For uniformly distributed values of x the rounded approximation x^* produced by this process has an average magnitude slightly larger than that of x, the amount of the difference of the averages of the true numbers and their rounded approximations being one-half unit in the most significant digit dropped. The chopping process produces approximate values with average magnitudes less than those of x, the amount of the difference in average values for uniformly distributed numbers being about one-half unit in the last digit kept. The more complicated rounding process commonly used by people produces a rounded approximation for which the average of the magnitudes of x^* and x are equal. The result of this rounding process differs from that for computers only when

$$|x - x^*| = (\tfrac{1}{2})b^p$$

in which case the computer rounded values is $|x^*| = |x| + (\tfrac{1}{2})b^p$; and in the more complicated rounding $|x^*| = |x| \pm (\tfrac{1}{2})b^p$, the sign being chosen so that the last digit of x^* is even.

In many computations using a single-address computer, a smaller region of uncertainty for the result can be obtained by rounding after a sequence of arithmetic operations rather than in each operation without any loss in computation time.

10-4. Generation and Propagation of Error

In automatic computations, digital approximations are used for numbers such as π, e, $\frac{1}{3}$, etc. The computer operations such as multiply,

[1] F. B. Hildebrand, "Introduction to Numerical Analysis," p. 8, McGraw-Hill Book Company, Inc., New York, 1956.

divide, floating point add, and shifts introduce additional errors into the mathematical computation due to the limitation on the number of digits. Following Householder,[1] the computer operation approximating a given mathematical operation ω will be called a pseudo operation and will be denoted by ω^*. The pseudo operation is defined by the particular computer and is defined therefore only for numbers of a particular digital form. Thus the desired mathematical result may be $x\omega y$ while the result produced by the computer is $x^*\omega^* y^*$, where x^* and y^* are the digital approximations for x and y. The error in this result is then

$$x\omega y - x^*\omega^* y^*$$

This error can be decomposed in several ways, a particularly useful decomposition being

$$(x\omega y - x^*\omega y^*) + (x^*\omega y^* - x^*\omega^* y^*)$$

The quantity in the first parentheses is the error that would be propagated if the operation were exact. The second parentheses contains the error generated by the pseudo operation ω^*.

These errors may be of the same or differing signs. In either case the magnitude of total error is not greater than the sum of the magnitudes of these two errors, i.e., $|x\omega y - x^*\omega^* y^*| \leq |x\omega y - x^*\omega y^*| + |x^*\omega y^* - x^*\omega^* y^*|$.

For illustration, consider the multiplication of two fixed point numbers on a 10-digit decimal computer, the decimal point being at the left of the first digit in the multiplier x, multiplicand y, and product z. In the computer the digital approximations

$$x^* = \pm .x\ xxx\ xxx\ xxx \qquad y^* = \pm .y\ yyy\ yyy\ yyy$$

are multiplied together and rounded to form the pseudo product

$$z^* = \pm .z\ zzz\ zzz\ zzz$$

Here subscripts for the digits in the representation are understood but not included for convenience only. An upper bound for the error $z - z^* = xy - x^* \cdot^* y^*$, where $\cdot^*$ denotes computer multiplication, can be obtained from

$$\begin{aligned} xy - x^* \cdot^* y^* &= xy - x^*y + x^*y - x^*y^* + x^*y^* - x^* \cdot^* y^* \\ &= (x - x^*)y + (y - y^*)x^* + (x^*y^* - x^* \cdot^* y^*) \end{aligned}$$

If x^* and y^* are correctly rounded digital approximations for x and y,

[1] A. S. Householder, "Principles of Numerical Analysis," McGraw-Hill Book Company, Inc., New York, 1953.

then $|x - x^*| \leq 5 \times 10^{-11}$, $|y - y^*| \leq 5 \times 10^{-11}$. The computer rounded product $x^* \cdot^* y^*$ differs not more than 5×10^{-11} from x^*y^*. Thus the error $|z - z^*|$ does not exceed 1.5×10^{-10}.

In fixed point addition the sum of the digital approximations is exact providing that the partial sums are in range. The magnitude of the computer sum for the digital approximations $x^* = .x\ xxx\ xxx\ xxx$, $y^* = .y\ yyy\ yyy\ yyy$ for two numbers x,y will then differ by less than 10^{-10} from the absolute value of the true sum $x + y$. If $-1 < x^* + y^* < 1$, the approximate sum is in range; otherwise the sum may exceed the capacity of the machine, and this overflow must be considered. In floating point computations the computer sum of two digital approximations in floating point notation may differ from the true arithmetic sum of these approximations. For nonzero numbers in the decimal floating point form

$$x^* = \pm, e_1e_2, xxxx\ xxxx = \pm(.xxxx\ xxxx)\ 10^{e_1e_2-50}$$
$$y^* = \pm, f_1f_2, yyyy\ yyyy = \pm(.yyyy\ yyyy)\ 10^{f_1f_2-50}$$

where $.1 \leq .xxxx\ xxxx, .yyyy\ yyyy < 1$, the sum of the numbers being approximated differs from the sum of the approximate numbers by

$$x + y - (x^* +^* y^*) = (x - x^*) + (y - y^*) + (x^* + y^*) - (x^* +^* y^*)$$

Since

$$|x - x^*| \leq 5 \cdot 10^{-8}|x^*|, |y - y^*| \leq 5 \cdot 10^{-8}|y^*|$$

for correctly rounded floating point numbers, and

$$|(x^* + y^*) - (x^* +^* y^*)| \leq 5 \cdot 10^{-8} \max\{|x^*|, |y^*|\}$$

it follows that the difference between the sum of the exact numbers and the sum of the approximate numbers does not exceed $15 \cdot 10^{-8} \cdot \max\{|x^*|, |y^*|\}$. The maximum relative error for the approximate sum is largest when x and y are of opposite signs and nearly the same magnitude. Some computers designed by government laboratories have had provision for automatically keeping track of the loss of precision in intermediate results.[1]

In general, an inspection of intermediate results in a floating point computation does not reveal as much information about loss of significant digits as is revealed by the intermediate results in a similar fixed point computation. That is, in fixed point operations, the accumulation

[1] For examples see the recent articles and reviews, in the *Journal* and the *Communications of the Association for Computing Machinery*, on range arithmetic and those on unnormalized floating point arithmetic.

zeros in the leading digit positions indicates a loss of significant digits; it in floating point operations, the leading precision digit is always non-zero unless the number is identically zero.

For a general function which has continuous derivatives, the Taylor series expansion can frequently be used in an error analysis. This series expansion is derived in most calculus texts. For a function of two variables $f(x,y)$, the Taylor series is

$$f(x,y) - f(x^*,y^*) = \xi \frac{\partial f(x^*,y^*)}{\partial x} + \eta \frac{\partial f(x^*,y^*)}{\partial y} + \frac{1}{2}\left(\xi^2 \frac{\partial^2 f}{\partial x^2} + 2\xi\eta \frac{\partial^2 f}{\partial x\, \partial y} + \eta^2 \frac{\partial^2 f}{\partial y^2}\right) + \frac{1}{6}\left(\xi^3 \frac{\partial^3 f}{\partial x^3} + 3\xi^2\eta \frac{\partial^3 f}{\partial x^2 \partial y} + 3\xi\eta^2 \frac{\partial^3 f}{\partial x \partial y^2} + \eta^3 \frac{\partial^3 f}{\partial y^3}\right) + \cdots$$

where $\xi = x - x^*$

$\eta = y - y^*$

When both ξ and η are small and the second and higher partial derivatives are not large, then

$$f(x,y) - f(x^*,y^*) \doteq \xi \frac{\partial f(x^*,y^*)}{\partial x} + \eta \frac{\partial f(x^*,y^*)}{\partial y}$$

For multiplication, $f(x,y) = xy$, two second partial derivatives are zero, and the complete Taylor series expansion is

$$xy - x^*y^* = \xi y^* + \eta x^* + \xi\eta$$

which may be easily verified by substitution.

For division, $f(x,y) = x/y$ and when ξ and η are small when compared with y

$$\frac{x}{y} - \frac{x^*}{y^*} \doteq \frac{\xi}{y^*} - \frac{\eta x^*}{y^{*2}}$$

This approximation error can be expressed exactly as

$$\frac{x}{y} - \frac{x^*}{y^*} = \frac{\xi\bar{y} - \eta\bar{x}}{\bar{y}^2}$$

where $\bar{x} = x^* + t\xi$ and $\bar{y} = y^* + t\eta$ and t is a number in the interval $0 < t < 1$.

When two or more operations are involved in a calculation, the sequence of operations may effect the error in the result of the computation. This fact is particularly evident from the examination of the fixed point calculation of xy/z. Since the result of a computer product of two numbers is

independent of which factor is the multiplier, this quantity xy/z can be approximated by one of the following three numbers which, in general, will not be equal

$$(x^* \cdot^* y^*) \div^* z^* \qquad (x^* \div^* z^*) \cdot^* y^* \qquad (y^* \div^* z^*) \cdot^* x^*$$

An analysis using the same procedures as in the preceding discussion for one operation can be used to derive error bounds for these three approximations for xy/z. The reader is referred to Chap. 1 of Householder's "Principles of Numerical Analysis"[1] for a detailed discussion of the error bounds for this and other short computations and a complete error analysis for a square-root computation and the computation of sin x and cos x.

10-5. Function Approximation

Since a significant percentage of computations involve one or more mathematical functions, a discussion of function approximation may be considered a desirable part of a book on programming. The computation steps performed by most digital computers are the basic operations of addition, subtraction, multiplication, division, and rounding of digital numbers. The first four of these operations are called the rational operations of algebra. Some computers have an operation which approximates the square root of a number (an irrational algebraic operation). Very few computers have operations for the approximation of other irrational algebraic functions or of the transcendental functions of analysis.

In nonautomatic computations, tables are used for evaluation of functions. This procedure may not be practical for automatic computation because of memory requirement for the table, time to read the table, or computation time required to find the correct argument in the table. In automatic computations, functions are evaluated by several techniques, each evaluation involving a sequence of computer operations. As far as the computer is concerned, these can be viewed as rounded rational approximations. Mathematically, the approximation techniques for function evaluation are quite distinct and can be separated into categories including: iterative procedures such as successive approximations, polynomial approximations, rational approximations, and table look-up. When successive functional values are needed, a functional equation, e.g., a differential equation or recursion relation, may be used for calculating the successive values of the function.

Polynomial approximations for functions have received considerable attention by mathematicians. Chebyshev polynomials are frequently

[1] *Op. cit.*

used to reduce the number of terms used in truncated power-series approximations.[1] Before the advent of automatic computation, however, little attention was given to the practical aspects of the approximation of functions by other rational expressions. Lately, however, this field has received some attention, several papers have been published, and the book "Approximations for Digital Computers" by C. Hastings[2] has received considerable attention by computer users. These approximations are often derived by semiempirical methods. The classical continued fractions frequently yield very good approximation formulas. Many of the approximations are published or reviewed in the journal *Mathematical Tables and Other Aids to Computation.*

The following example derived by H. J. Maehly illustrates the simplicity of a rational approximation for 2^x, a function frequently used in computation with a binary computer:

$$2^x \doteq \frac{1+T}{1-T} \qquad -\tfrac{1}{2} \leq x \leq \tfrac{1}{2}$$

where $T = c_1x + c_3x^3 + c_5x^5$

$c_1 = .17328\ 679471$

$c_3 = -.00173\ 448962$

$c_5 = .00002\ 072333$

The error in this approximation is less than 2×10^{-10}. A power-series approximation for 2^x requires nine terms for the same accuracy as the above rational approximation and would require about 25 per cent more computation time for each evaluation. In general, the approximation of a function by a truncated power series will not give as accurate an approximation as can be achieved by an approximation polynomial with the same number of terms. Similarly, a rational expression with the same number of terms may provide an even more accurate approximation.

The interest in approximation has increased to the extent that a mathematical meeting at the University of Wisconsin in the summer of 1958 was devoted entirely to function approximation.

10-6. Interpolation and Differencing

Since interpolation is one of the most basic and frequently used processes in numerical computation, it is essential for most programmers to

[1] M. V. Wilkes, D. J. Wheeler, and S. Gill, "Programs for an Electronic Digital Computer" (revised), Addison-Wesley Publishing Company, Reading, Mass., 1957.

[2] C. Hastings, Jr., "Approximations for Digital Computers," Princeton University Press, Princeton, N.J., 1955.

be acquainted with some of the general theories of interpolation. In this section, however, only the use of polynomial interpolation is discussed and a few formulas are reviewed. Interpolation is used in automatic computation for function approximation and checking results, as a part of many numerical procedures such as procedures for finding solutions to equations and for finding the maximum of a function, and for developing numerical procedures such as numerical integration formulas and procedures for obtaining approximate solutions to differential equations. In processing numerical data, interpolation may be used to average, i.e., smooth, errors in the data.

The basis of interpolation is the approximation of functions by combinations of appropriate simpler functions. Polynomials, rational functions, trigonometric functions, and exponential functions are the most frequently used approximation functions. Polynomial and rational functions are particularly well adapted for computation, although in some cases they may not be suitable.

Differences of functional values occur in polynomial interpolation formulas, and it is desirable to have a notation for these differences. Let $f(x)$ be a function of a real variable x. The *first-order difference* of $f(x)$ is the difference of two function values, i.e., $f(a_1) - f(a_0)$. The *first-order divided difference*, $f(a_1,a_0)$, for $f(x)$ is

$$f(a_1,a_0) = \frac{f(a_1) - f(a_0)}{a_1 - a_0}$$

The second-order difference for $f(x)$ is the difference of two contiguous first-order differences, e.g., $f(a_2) - 2f(a_1) + f(a_0)$ (this difference is particularly useful when $a_2 - a_1 = a_1 - a_0$) and the second-order divided difference is

$$f(a_2,a_1,a_0) = \frac{f(a_2,a_1) - f(a_1,a_0)}{a_2 - a_0}$$

similarly, the third-order divided difference for $f(x)$ is

$$f(a_3,a_2,a_1,a_0) = \frac{f(a_3,a_2,a_1) - f(a_2,a_1,a_0)}{a_3 - a_0}$$

and so on.

The divided difference of a linear combination of two functions is equal to the same linear combination of the corresponding divided difference of each function. The nth divided difference of a polynomial of degree n is constant and the $(n + 1)$th divided difference is zero.

The Newton divided difference interpolation formula for an nth-degree polynomial approximation $p(x)$ to the function $f(x)$ is

$$\begin{aligned} p(x) = f(a_0) &+ (x - a_0)f(a_1,a_0) + (x - a_0)(x - a_1)f(a_2,a_1,a_0) \\ &+ (x - a_0)(x - a_1)(x - a_2)f(a_3,a_2,a_1,a_0) + \cdots \\ &+ (x - a_0)(x - a_1) \cdots (x - a_{n-1})f(a_n, a_{n-1}, \ldots, a_1, a_0) \end{aligned}$$

This interpolation polynomial $p(x)$ has the same values as $f(x)$ when $x = a_0, a_1, \ldots, a_n$.

As Booth illustrates in his book *Numerical Methods*,[1] polynomial interpolation must be used with caution. Erroneous results may be obtained when the high-order differences used in the interpolation formula are large.

The above interpolation is applicable for either direct or inverse interpolation as the arguments need not be equally spaced. It can also be used to find an approximation for a maximum occurring between tabular values. Here the maximum of the interpolation polynomial is used for the approximation of the function maximum.

Divided differences are frequently used to detect errors in tables of results. In this check the divided differences are checked for an abrupt change caused by the error in the table. A smooth function, i.e., one with slowly changing derivatives, will have slowly changing divided differences. An error in one function value will affect adjacent differences in opposite directions and, in general, the error will appear as a larger relative change in the differences than in the function.

10-7. Iterative Methods for Solving Equations

In many cases iterative procedures for calculation of values defined by a functional relation are computationally much simpler than direct methods. In scientific computations this is particularly evident. Thus there is considerable practical interest in iterative procedures. Iterative methods are also used to reduce the error in results obtained by direct methods. A large number of iterative procedures for solving linear and nonlinear equations and systems of equations are described in the literature. Books on numerical methods such as Householder's "Principles of Numerical Analysis" and Hildebrand's "Introduction to Numerical Analysis" describe some of these methods in detail and indicate where additional methods are described in mathematical journals. Two other bibliographies by A. S. Householder on numerical analysis may be found

[1] A. D. Booth, "Numerical Methods," Academic Press, Inc., New York; Butterworth's Scientific Publications, London, 1955.

in the *Journal of the Association for Computing Machinery*, vol. 3, pp. 85–100, 1956, and the report, ORNL-1897, published by Oak Ridge National Laboratory. In the remainder of this section some aspects of iterative procedures for determining the square root of a number are discussed to supplement the square-root example previously considered.

A general class of iterative procedures for solving the equation $f(x) = 0$ is obtained by reformulating the relation to the form $x = F(x)$. If the sequence x_0, $x_1 = F(x_0)$, $x_2 = F(x_1)$, . . . , $x_{n+1} = F(x_n)$ converges to a finite limit X, then this limit is, in general, a solution to the equation $x = F(x)$. This process is illustrated in Fig. 10-7-1. These figures contain the two graphs $y = x$ and $y = F(x)$ for two examples. The sequence

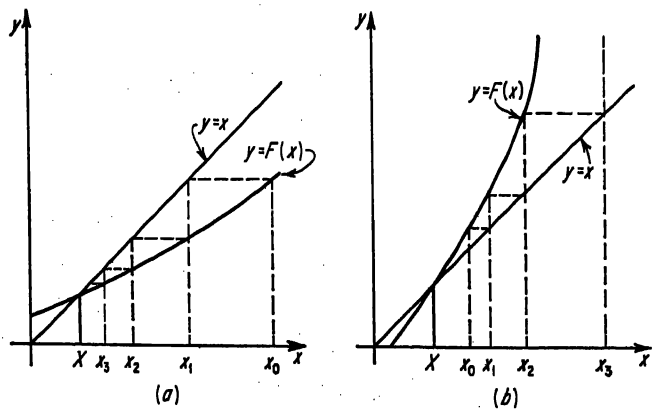


FIG. 10-7-1. Illustrations of an iterative process for $x_{n+1} = F(x_n)$.

x_0, x_1, x_2, . . . , is found by successive evaluation of the function $F(x)$ as shown by the dotted lines. Part (*a*) of Fig. 10-7-1 demonstrates an example where the sequence converges, and part (*b*) demonstrates an example of divergence. Other examples could be constructed for cases where convergence or divergence of the sequence would depend on the value of the initial approximation x_0. Clearly, the usefulness of such an iterative procedure depends on the nature and speed of the convergence of the successive iterates. It is also of import in computation that practical estimates for the solution error be known in terms of the successive iterates, x_n, x_{n+1}, ... From Fig. 10-7-1(*a*) and (*b*) it is evident that a *sufficient* condition for convergence of the sequence is that

$$\left| \frac{F(x) - F(X)}{x - X} \right| < 1$$

for all x in the iteration region. This condition is satisfied if the magnitude of the derivative, dF/dx, is less than 1 for all x in the iteration region. By definition $F(x_n) = x_{n+1}$ and $F(X) = X$, thus the preceding sufficient condition may be written in the form

$$|x_{n+1} - X| < |x_n - X|$$

The determination of the square root of a number by an iterative process was discussed in Chaps. 1 and 3. An equation for the square root x of a number a may be written as $x^2 - a = 0$. This equation can be reformulated to the form $x = F(x)$ in many ways. Three examples are

(a) $$x = \frac{a}{x}$$

obtained by dividing $(x^2 - a = 0)$ by x;

(b) $$x = \tfrac{1}{2}\left(x + \frac{a}{x}\right)$$

obtained by writing $x^2 = a$ as $2x^2 = x^2 + a$ and dividing by $2x$; and

(c) $$x = \frac{x(3a - x^2)}{2a}$$

obtained by writing $0 = a - x^2$ as $2ax = 3ax - x^3$ and dividing by $2a$. The sequence x_0, $x_1 = F(x_0)$, $x_2 = F(x_1)$, . . . does not converge for the form (a), but converges for the remaining two forms. Form (c) has computational advantages for a computer without a division operation. It is of interest to examine the value of $\dfrac{F(x) - F(X)}{x - X}$ for the last two forms. If

$$F(x) = \tfrac{1}{2}\left(x + \frac{a}{x}\right)$$

then $$\frac{F(x) - F(X)}{x - X} = \frac{F(x) - F(\sqrt{a})}{x - \sqrt{a}} = \frac{x - \sqrt{a}}{2x}$$

This quantity has a magnitude less than 1 when $x > \sqrt{a}/3$. Thus, if x_0 is greater than $\sqrt{a}/3$, then $|x_0 - \sqrt{a}| > |x_1 - \sqrt{a}| > |x_2 - \sqrt{a}|$, etc. Also, since $F(x_n) = x_{n+1}$ for any $x_n > 0$

$$\frac{x_{n+1} - \sqrt{a}}{x_n - \sqrt{a}} = \frac{x_n - \sqrt{a}}{2x_n}$$

or $x_{n+1} - \sqrt{a} = \frac{1}{2x_n}(x_n - \sqrt{a})^2$ a positive quantity and $x_1 - \sqrt{a} > x_2 - \sqrt{a} > x_3 - \sqrt{a} > \cdots > 0$. Therefore the successive iterates (except perhaps the initial approximation x_0) are all greater than the square root and form a sequence which converges to $\sqrt{a}$ monotonically from above. The sequence converges rapidly.

For form (*c*), $F(x) = x(3a - x^2)/2a$; thus

$$\frac{F(x) - F(X)}{x - X} = \frac{F(x) - F(\sqrt{a})}{x - \sqrt{a}} = -\frac{(x - \sqrt{a})(x + 2\sqrt{a})}{2a}$$

and

$$x_{n+1} - \sqrt{a} = -\frac{x_n + 2\sqrt{a}}{2a}(x_n - \sqrt{a})^2$$

Comparing this with the similar expression for the form (*b*), the comparison of the convergence rate for the two methods depends on the relative size of the factors $(x_n + 2\sqrt{a})/2a$ and $1/2x_n$. When x_n is near $\sqrt{a}$, this first factor is about $\frac{3}{2\sqrt{a}}$ and the second factor about $\frac{1}{2\sqrt{a}}$. Thus, the iteration for form (*b*) converges somewhat faster than the iteration for form (*c*). The sequence of iterates for form (*c*) converges to $\sqrt{a}$ when $0 < x < \frac{\sqrt{a}}{2}(\sqrt{17} - 1)$. For x_0 in this region the iterates $x_1, x_2, \ldots,$ converge monotonically to $\sqrt{a}$ from below.

The form (*b*) above can be derived by Newton's method. This method is applicable for the solution of equations and has recently been extended and applied to functional equations including nonlinear integral equations.

10-8. Bibliography

The following bibliography is a representative list of recent books on numerical analysis and methods. The reference included with each book is that of a review in the *Mathematical Reviews*. These reviews may be used as a guide for books to acquire for a computation library.

Allen, D. N. deG.: "Relaxation Methods," McGraw-Hill Book Company, Inc., New York, 1954. *Math. Revs.*, vol. 15, p. 831.

Bodewig, E.: "Matrix Calculus," North-Holland Publishing Company, Amsterdam, 1956. *Math. Revs.*, vol. 18, p. 235.

Booth, Andrew D.: "Numerical Methods," Academic Press, Inc., New York; Butterworths Scientific Publications, London, 1955. *Math. Revs.*, vol. 16, p. 861.

Buckingham, R. A.: "Numerical Methods," Pitman Publishing Corporation, New York, 1957.

Collatz, L.: Numerische und graphische Methoden, "Handbuch der Physik," Bd II, Springer-Verlag OHG, Berlin, Vienna, 1955. *Math. Revs.*, vol. 18, p. 71.

Collatz, L.: "Numerische Behandlung von Differentialgliechungen," Springer-Verlag OHG, Berlin, Vienna, 1955. *Math. Revs.*, vol. 16, p. 962.

Collatz, L.: "The Numerical Treatment of Differential Equations," 3d ed., Springer-Verlag OHG, Berlin, Vienna, 1960. *Math. Revs.*, vol. 22, p. 57.

Crandall, Stephen H.: "Engineering Analysis; A Survey of Numerical Procedures," McGraw-Hill Book Company, Inc., New York, 1956. *Math. Revs.*, vol. 18, p. 674.

Dwyer, Paul S.: "Linear Computations," John Wiley & Sons, Inc., New York; Chapman and Hall, Ltd., London, 1951. *Math. Revs.*, vol. 13, p. 283.

Forsythe, George E., and Wolfgang R. Wasow: "Finite-difference Methods for Partial Differential Equations," John Wiley & Sons, Inc., New York, 1960.

Grossman, Walter: "Grundzüge der Ansgleichungrsrechnung nach der Methode der kleinstein Quadrate nebst Anwendungen in der Geodäsie," Springer-Verlag OHG, Berlin, Vienna, 1953. *Math. Revs.*, vol. 15, p. 650.

Hartree, D. R.: "Numerical Analysis," Oxford University Press, New York, 1952. *Math. Revs.*, vol. 14, p. 690; vol. 20, p. 1111.

Hastings, Cecil, Jr.: "Approximations for Digital Computers," Princeton University Press, Princeton, N.J., 1955. *Math. Revs.*, vol. 16, p. 963.

Hildebrand, F. B.: "Introduction to Numerical Analysis," McGraw-Hill Book Company, Inc., New York, 1956. *Math. Revs.*, vol. 17, p. 788.

Householder, Alston S.: "Principles of Numerical Analysis," McGraw-Hill Book Company, Inc., New York, 1953. *Math. Revs.*, vol. 15, p. 470.

Kantorovich, L. V., and V. I. Krylov: "Approximate Methods of Higher Analysis," Interscience Publishers, Inc., New York, 1958. *Math. Revs.*, vol. 21, p. 982.

Kopal, Zdeněk: "Numerical Analysis, with Emphasis on the Application of Numerical Techniques to Problems of Infinitesimal Calculus in Single Variable," John Wiley & Sons, Inc., New York, 1955. *Math. Revs.*, vol. 17, p. 1007.

Kunz, Kaiser S.: "Numerical Analysis," McGraw-Hill Book Company, Inc., New York, 1957. *Math. Revs.*, vol. 19, p. 460.

Lanczos, Cornelius: "Applied Analysis," Prentice Hall, Inc., Englewood Cliffs, N.J., 1956. *Math. Revs.*, vol. 18, p. 823.

Lukaszewicz, Józef, and Mieczyslaw Warmus: "Methody Numeryczne i Graficzne," Cześć I ("Numerical and Graphical Methods," part I), Państwowe Wydawnictwo, Warsaw, 1956. *Math. Revs.*, vol. 18, p. 235.

Mikeladze, Š. E.: "Cislennye Metody Matematiceskogo Analiza" ("Numerical Methods of Mathematical Analysis"), Gosudarstv, Izdat, Tehn-Teor, Ltd., Moscow, 1953. *Math. Revs.*, vol. 16, p. 627.

Milne, William Edmund: "Numerical Solution of Differential Equations," John Wiley & Sons, Inc., New York; Chapman and Hall, Ltd., London, 1953. *Math. Revs.*, vol. 16, p. 864.

Mineur, Henri: "Techniques de calcul numérique a l'usage des mathématiciens, astronomes, physiciens, et ingénieurs, suivi de quatre notes par Mme. Henri Berthod-Zuborowsk, Jean Bouzitat, et Marcel Mayot," Librairie Polytechnique Charles Béranger, Paris, 1952. *Math. Revs.*, vol. 15, p. 557.

Nielsen, Kaj L.: "Methods in Numerical Analysis," The Macmillan Company, New York, 1956. *Math. Revs.*, vol. 17, p. 897.

"Proceedings of Symposia in Applied Mathematics," vol. 6, "Numerical Analysis," McGraw-Hill Book Company, Inc., New York, 1956. *Math. Revs.*, vol. 17, p. 1241; vol. 18, pp. 71, 73, 74, 824, 825; vol. 19, pp. 179, 180.

Ralston, Anthony, and Herbert S. Wilf: "Mathematical Methods for Digital Computers," John Wiley & Sons, Inc., New York, 1960.

Richtmyer, R. D.: "Difference Methods for Initial-value Problems," Interscience Publishers, Inc., New York, 1957. *Math. Revs.*, vol. 20, p. 438.

Salvadori, Mario G.: "Numerical Methods in Engineering, with a Collection of Problems by Melvin L. Baron," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1952. *Math. Revs.*, vol. 16, p. 1154.

Scarborough, James B.: "Numerical Mathematical Analysis," 3d ed., John Hopkins Press, Baltimore; Oxford University Press, London, 1955. *Math. Revs.*, vol. 17, p. 789; vol. 20, p. 68.

Shaw, F. S.: "An Introduction to Relaxation Methods," Dover Publications, Inc., New York, 1953. *Math. Revs.*, vol. 15, p. 353.

Southwell, R. V.: Vol. I, "Relaxation Methods in Engineering Science," 1940; Vol. II, "Relaxation Methods in Theoretical Physics," 1946; Vol. III, "Relaxation Methods in Theoretical Physics, a Continuation of the Treatise Relaxation Methods in Engineering Science," 1956, Oxford University Press, London. *Math. Revs.*, vol. 3, p. 152; vol. 8, p. 355; vol. 18, p. 677.

"Symposium on Monte Carlo Methods," University of Florida, 1954, John Wiley & Sons, Inc., New York; Chapman and Hall, Ltd., London, 1956. *Math. Revs.*, vol. 17, p. 1241; vol. 18, pp. 72, 151, 152, and 153.

"Tables of Chebyshev Polynomials $S_n(x)$ and $C_n(x)$," National Bureau of Standards Applied Mathematics Series, no. 9, Washington, D.C., 1952. *Math. Revs.*, vol. 16, p. 959.

Zürmuhl, R.: "Praktische Mathematik für Ingenieure und Physiker," Springer-Verlag OHG, Berlin, Vienna, 1953. *Math. Revs.*, vol. 15, p. 470.

11

ORGANIZATION OF A COMPUTER INSTALLATION

11-1. Introduction

In computer installations used primarily for accounting, the major effort in programming and coding is often performed before the acquisition of the computer. Such installations have regularly scheduled operations. For example, the computer may generate certain payroll reports once a year, others semiannually, others once a month, others semimonthly, and still others each week. The same computer may be used to produce for each department of the company a similar set of budget reports, inventory reports, etc. Thus, a specified schedule for processing each report will be assigned for the computer. Enough of the aforementioned routines are programmed, coded, and checked before the computer is installed so that these routines may be used in the acceptance tests of the computer and so that the computer can justify its existence immediately after it has been installed. As soon as possible the remaining routines are added to the schedule of the computer. Except for improving existing routines, changing existing routines to meet changes in the policies of the company, and the writing of routines for special reports, the major programming and coding effort for accounting installations occurs during the early life of the computer and may be considered as a part of the initial cost of the installation.

When the computer running time for a problem is large due to the repeated processing of new input information, the problem is referred to as a *production* problem. Usually, the ratio of man-hours of programming time to computer running time for production problems is not much greater than one to one for medium-speed computers. Examples of production problems are those associated with calculations for payroll,

inventory, engineering design, market analysis, and routine survey problems in the social and political sciences.

In computer installations used for scientific computation, many problems require less computing time than the time required to program and code the problem. Often, the ratio of man-hours of programming time to computer time for a problem is greater than forty to one. Furthermore, most of the programming is done as the problems arise, and only a small amount may be done before the computer is installed. For such an installation, more than 20 programmers may be required to keep a medium-speed computer busy 8 hours per day.

For a particular computer installation, the programming requirements after the computer is installed may fall somewhere between the limits for the two types of installations described above. In general, the amount of programming time required after installation is inversely related to the ratio of computer time required for production problems to total computer time used for problems.

A description of the organization of and operation of a computer installation in which the ratio of programming time to computer time is large displays the organizational and operational problems of most computer installations. These organizational and operational problems usually revolve around the economics of the computer. That is, since the computer is usually an expensive piece of equipment, one attempts to use the computer as much as possible for productive work by being sure that the problems to be computed are properly stated, that the computer schedule is efficient, and that program debugging procedures use a minimum of computer time. In this chapter we will discuss possible organizations for scientific computer installations, some reasonable procedures for debugging programs, and good operational procedures for the computer.

11-2. Scientific Automatic Digital Computer Installation

An automatic digital computer for scientific computation may be located in an applied mathematics and statistics department. Such a department may include one or more of the following groups: an analysis group, a statistics group, a hand computing group, an analog computer group, and an automatic digital computer group. The applied mathematics and statistics department may be part of a mathematics division or some other division. The place of an automatic digital computer group within the organization of a university, a company, or a laboratory will depend to a large extent upon the administrative policies of the particular governing body which established the computing center.

A desirable mode of operation is to have problems first considered by the analysis group or statistics group to determine that they are properly stated and to determine whether or not the only available and appropriate method of solution is by computational methods. If computational methods are required, the group should also determine the most desirable method of computation for the problem, i.e., the numerical method and the mathematical statement of the problem for that numerical method.

Let us assume that a problem has been determined to be one of the class which is appropriate for an automatic digital computer. Then the analysis or statistics group should, in addition to stating the problem mathematically for the computer, determine the accuracy of the numerical answers and the computational methods for determining this accuracy and also specify numerical tests to be used in checking the routine as coded for the computer. After the problem has been programmed, coded, and run on the computer, the answers should be returned to the analysis or statistics group so that they may be checked for errors. In this manner additional reliability may be placed on the answers for the originator of the problem.

After the problem has been programmed and coded and examined for programming and coding errors, the programmer should make a list of instructions for debugging the routine on the computer. This list should be sufficient in detail that a computer operator not familiar with the problem may run the problem on the computer. This list of instructions should be accompanied by a list of estimated computer running times for each phase of the problem. After the routine has been debugged, a correct list of instructions for the routine should be collected along with the actual computer running times for each phase of the problem.

The preceding remarks are equally valid whether one person carries the problem all of the way from the analysis through the running of the problem on the computer, or whether different people are involved for each phase of the work.

Several existing computer installations follow the organization described above, and all problems processed on a digital computer must enter the organization through the analysis group and be processed by the mathematics department. Such processing of problems is usually referred to as a "closed shop" operation of the digital computer. For economic, administrative, or other reasons, scientific computing installations often are not attached to such a complete mathematics and statistics department. For example, the value attached to a numerical solution may not justify the cost of a complete investigation, or the prob-

lem originator may be able to determine from the answers whether or not the problem has been run successfully. Often, the problem originator desires complete control over all phases or certain of the phases of the work and wishes to assume responsibility for this part of the work. Thus, many computer installations are part of an organization which offers assistance in mathematics, statistics, programming, coding, and computer operation to its customers and allows them to choose the assistance they desire. Such installations are usually referred to as having "open shop" operation. Obviously, under such operation the answers to some problems will be computed which have already been or could be solved by more suitable methods and other problems will be computed incorrectly. Such occurrences are usually justified on an economical basis and attention is drawn to the fact that open shop operation may afford some of the benefits of closed shop operation when desired by the problem originator.

11-3. Records of Computer Problems

By specifying the contents of the file that should be kept for each problem solved on a digital computer by the mathematics and statistics department, many of the procedures for programming and coding will become apparent. Before stating how records should be kept, we point to the necessity of keeping accurate and intelligible records. Often after a problem as originally specified has been completed, the originator desires further calculations based on changes of certain parameters in the problem. If adequate records have been kept, then one may determine if the existing routine may be used with the new parameters or, because the existing scaling of the problem does not permit calculations using parameters in the range of the new set, one may determine which parts of the routine will have to be recoded. Often the detailed analysis of answers requires months of effort, and it is not until some time after the computing has been completed that an error is detected. After the detection of such an error, it is desirable to determine if the cause was mathematical, programming, or computational before rerunning that part of the problem in which the incorrect answer was calculated. Again, an adequate record of the problem may be advantageous. Often some of the computational procedures involved in a problem to be calculated have already been programmed as a part of another problem. Thus, an adequate record of the previous problem may save duplication of effort. We assume that these brief remarks are sufficient indication of the neces-

sity for maintaining adequate records of problems solved with the aid of a digital computer and now turn to what should be included in these records.

Let us assume that the problem entered the mathematics and statistics department through the statistics or mathematics analysis group. In one of these two groups the problem was stated mathematically, and it was determined that the solution should be by numerical procedures on an automatic digital computer. It was then analyzed to determine what numerical procedures would be involved. The problem was then restated in numerical analysis form for the ranges of variables and parameters desired. As a result of this work the problem file should contain a mathematical or statistical statement of the problem including the ranges of variables and parameters to be used by the computer. This part of the file should include the name of the person who performed this work and the date of completion. By identifying the person who did the analysis a source of knowledge is located for future problems of a similar type or for future discussions involving the same problem. The date is necessary so that one may determine if better methods of solution have been developed since the problem was analyzed.

The statement of the problem by the analyst for the computer was next used to construct the detailed flow chart and coding for the problem. After the code has been debugged on the computer, the problem file should contain a correct copy of the flow chart, coding, instructions for operating the computer, the names of the persons who performed this work, and the date the programming and coding was completed. The names are helpful since it is desirable to have the person who did the original work make changes when the problem is changed. The date is necessary since computers may undergo alterations which imply changes in command codes, or since changes may be made for the subroutines used. Thus a routine coded before a certain date may no longer run on the computer until parts of its code have been brought up to date.

After the routine has been debugged and the problem has been run on the computer, the problem file should contain a copy of input data and answers for a typical set of parameters. This information may be used as a computer checking routine for the program, data, and operation procedures for that particular problem for future runs. Also, a copy of the program on punched cards, on punched paper tape, or on magnetic tape should be stored for future use. Since this type of information is usually stored separately, a cross reference to its location should become part of the problem file.

Finally, for those problems which may be used as subroutines, abbrevi-

ated records, as already outlined in the chapter on subroutines, should be made for the library of subroutine abstract descriptions.

11-4. Programming and Coding

As already indicated in the previous section a correct copy of the flow diagram and coding is a part of the problem file. The file is constructed for reference purposes and often will be used by persons not originally associated with the problem. Thus, the flow diagrams and code should be in intelligible form. One way of having flow diagrams and coding in intelligible form is to have all members of the programming staff use the same rules for coding. A major portion of this book has been devoted to exposing the reader to a systematic set of rules for constructing each.

Aside from providing the latest (and most correct) version, a reason for inserting the flow diagram and coding in the problem file after the routine has been debugged is that either or both may originally contain subroutines, the sole purpose of which is to assist the programmer in debugging, and these debugging routines may be eliminated from the problem file leaving a clearer description of the problem. The manner in which the debugging routines are inserted and how they are used will be discussed in the next section.

11-5. Debugging of Routines

A useful, albeit time-consuming, routine for automatic digital computers is a *trace* routine. Such a routine is used to record the performance of another routine. The trace routine is supplied with beginning-of-trace and end-of-trace memory locations for the routine to be tested, and it follows and records the operations of the other routine from the designated beginning-of-trace to the designated end-of-trace location. The trace routine lists the command at the beginning-of-trace location and lists the data required by that command. The computer, under the control of the trace routine, then performs that command and the trace routine lists the result. The next command of the routine being tested is then executed under the control of the trace routine. Again, the data required by that command and the results for that computer operation (e.g., contents of the relevant arithmetic and control registers) are listed by the trace routine. Similarly, the trace routine follows each command of the routine being tested and lists it and the pertinent information until the command located at the end-of-trace memory location has been traced. There are several ways in which a trace routine may be used in

debugging. We will discuss one of these after we have exhibited several general rules for debugging.

Trace routines are often modified so that they do not record each command and the data associated with it but record only a subset of these commands and the data required by them. This subset might consist of all commands of a single type (i.e., all transfer commands or all store commands) or it might consist of all commands which have been specially tagged (i.e., since the M_1 address is ignored in the three-address command UCD, a specially tagged command might be those UCD commands with $M_1 = 001$). Such trace routines are referred to as *selective trace routines.*

We remind the reader that the large high-speed automatic digital computers are expensive to operate. We will discuss debugging procedures for these computers since they may also be used for medium and small automatic digital computers. The reasoning is that the programmer should learn habits which do not have to be changed as he switches from one computer to another. A habit the programmer should develop, insofar as debugging is concerned, is that the debugging should be performed, as much as possible, automatically by the computer and without intervention by the operator or the programmer. Such intervention is usually time-consuming and is susceptible to human error.

Although many computers have special commands or console switches designed to aid the programmer in debugging routines, these commands vary from one computer to another. We will assume no special command or switches for this purpose. First, let us look at the mechanics of testing a computer routine. For this, we assume a check problem exists for the routine to be debugged and that intermediate results have been calculated at desired points in the problem. Let these results be designated by p_i, $i = 1, 2, \ldots, n$. The computer is expected to calculate a set of intermediate results $P_i = p_i + \epsilon_i$ where the bounds E_i for the magnitude of the calculation errors for $|\epsilon_i|$ are known. Just after the calculations for each intermediate result P_i, an unconditional transfer command is inserted in the problem routine which transfers control to the next command of the routine. Designate these unconditional transfer commands by $U(\alpha_i)$, $1 \leq i \leq n$, where α_i is the address of that next command. When the problem routine is to be debugged, instead of entering the routine at its initial command, designated by α_0, it is entered through a debugging routine which replaces each $U(\alpha_i)$ by $U(\beta)$ and then transfers control to the command α_0. The essential part of the debugging routine is given in Fig. 11-5-1. Debugging routines such as these automatically check the intermediate values as the problem proceeds and

dumps only that part of the memory which is involved in an error. The debugging routine is furnished with a list of the p_i, E_i, and α_i and a list of $M(P_i)$ or extracts the $M(P_i)$ from the store command in cell $\alpha_i - 2$. When the debugging routine is initially loaded, $i = 1$.

An alternative procedure may be one that incorporates the trace routine. This debugging routine is the same as the preceding except

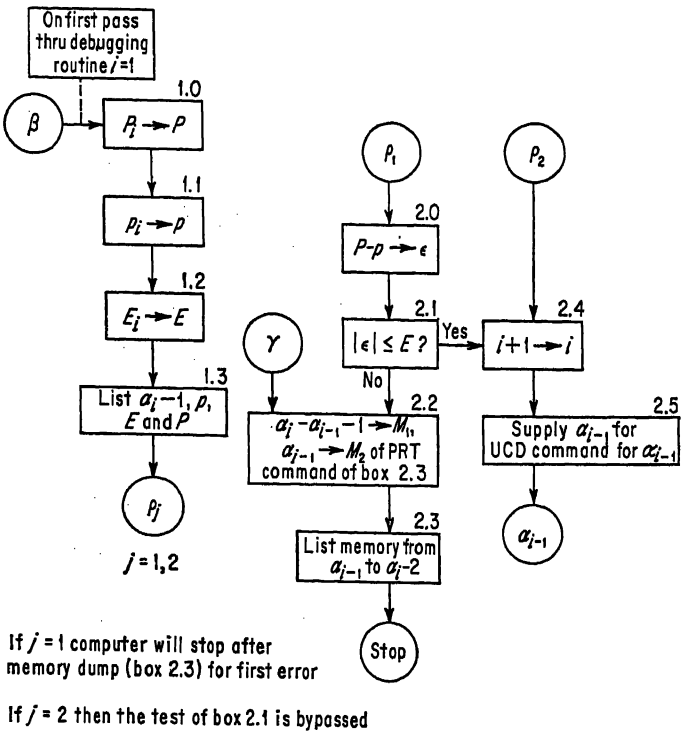


FIG. 11-5-1. A flow diagram for a debugging routine.

that the trace routine is used in place of the memory dump routine. Let δ be the location of the entry to the trace routine. The essential part of the flow diagram for the debugging routine using the trace routine is given in Fig. 11-5-2. After an error occurs, this routine causes the problem to be recalculated for the test problem up to the last correct intermediate result and then to be traced from there through the calculations which involve the error.

Sometimes, either the error bounds E_i, determined from a numerical

analysis of the problem, are so large as to be meaningless, or the problem is too complicated for the computation of a set of p_i to be practicable. In such cases, one must rely on the programmer to determine by an inspection of the P_i if the P_i have been properly calculated. By setting $\rho_j = \rho_2$ in the coding for the flow diagram of either Fig. 11-5-1 or Fig. 11-5-2, a complete listing of the partial answers will be obtained.

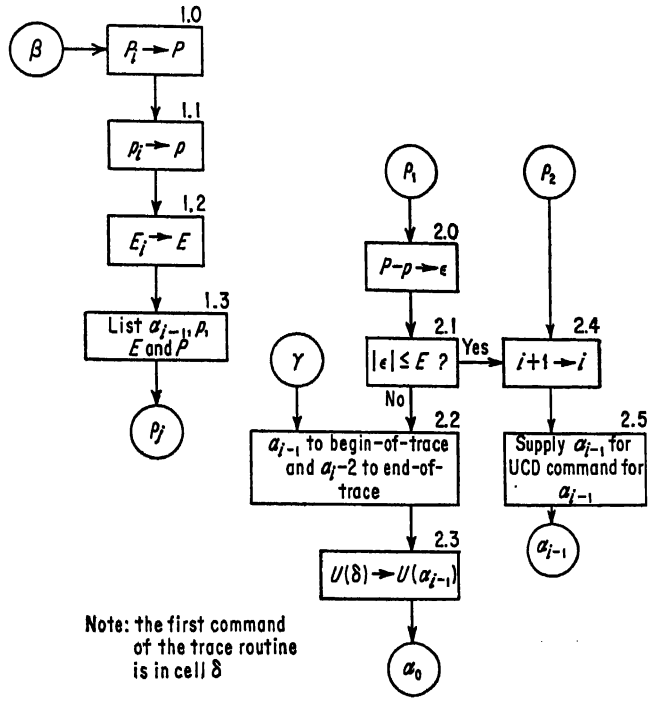


FIG. 11-5-2. Changes to flow diagram of Fig. 11-5-1 to incorporate a trace routine.

The preceding discussions exhibit the basic mechanism of testing a routine; however, they do not form a complete picture. Suppose, for example, the computer correctly calculates the intermediate result P_{i-1} but does not perform the calculation for P_i, i.e., it either stops because of an illegitimate command, an overflow, or some other error, or the computer performs a set of commands over and over without exiting from the set, i.e., the computer gets into a closed loop. Having time estimates for each phase of the problem enables one to determine if the computer is in a closed loop. For errors which cause the computer to calculate P_{i-1} correctly but not to reach P_i, the computer is stopped if it has not already

stopped. In debugging by the scheme of Fig. 11-5-1 (and Fig. 11-5-2), α_{i-1} and α_i are furnished for the debugging routine, the unconditional transfer command $U(\gamma)$ is placed in the control register of the computer, and the computer is restarted. In this manner, for Fig. 11-5-1 (and Fig. 11-5-2) the debugging routine is entered at γ, and that part of the memory is listed (that part of the memory is traced) in which the error has occurred.

The choice of intermediate results p_i of the check problem to be used for comparison with the intermediate results P_i of the computer should be made by the programmer to satisfy his needs. If the problem routine is compiled in part from the library of subroutines, then another set of intermediate results are the quantities used as inputs to the subroutines and the results of the subroutines.

Other types of debugging routines include "memory dump" and "address check." These routines are usually used after an unanticipated error has occurred and is recognized while a program is being processed on the computer. For example, the program (which had been successfully checked on the computer for the test case) is being used in the computation for a desired problem when an error is detected. Such an error might be recognized in several ways: an overflow might occur and cause the computer to stop, a result listed by the on-line listing unit might be an impossible answer for the problem, or the computer processing of the program might not be progressing according to its proper time schedule. Memory dump routines are designed to be located in any part of the memory. Thus, one memory dump routine may be used for many problems by locating the dump routine in a portion of the memory not used by the problem routine. Dump routines are usually flexible to the extent that they allow a choice of output units and a selection of formats for each output unit. For example, if the on-line printer is used, from one to eight consecutive words in the memory and the address of the first word may be listed per line. Since dumping all of the memory associated with a problem routine may require appreciable time, the dump routine is usually selective in nature, that is, it may be asked to dump the contents of arbitrary sections of the memory of varying length. Operationally, a simple way of accomplishing this is by designing the routine to request a beginning-of-memory and end-of-memory dump locations. After this information is supplied by the computer operator, the dump routine lists the contents of each consecutive memory location between the specified limits and then asks for a new set of limits. Thus, when an error occurs, the memory dump routine need only be used to list the contents of the memory locations associated with that error.

An error which occurs frequently enough that a special debugging routine has been designed for detection of its location is that of the computer trying to carry out an order located in a part of the memory not used by the problem routine. Since the unused parts of the memory are usually filled with STOP commands, such an error causes the computer to stop computation. Errors of this type are often traced to a computer error in the loading of the problem routine. The debugging routine used to track down the error in this case is usually called an "address check" routine. The address check routine is provided with the address of the erroneous STOP command and the memory locations of the commands of the problem. It then checks all transfer commands for those containing the address of the erroneous STOP command and lists the possible culprits and their addresses. Transfer commands whose transfer addresses are modified by the routine or by the contents of a *B*-box might not be detected by the address check routine. A selective trace routine which tests the program in a dynamic manner, however, would be capable of detecting the exact location of the error.

11-6. Operation of the Computer

Before a routine is debugged or run, the programmer should make a complete list of instructions for processing the routine on the computer. These instructions should be used by the computer operator or by the programmer if he runs the problem himself. They should include the manner in which the problem is loaded into the computer, the setting of switches on the console, and any special requirements for the listing, punching, or tape units. Computer running times should be given for each phase of the problem and a description of the listing should be given for each listing that will occur during the computation. Detailed instructions for restarting the problem with the last set of partial answers should be included for long problems. Similarly, instructions should be stated for dumping the necessary parts of the memory should the computation not be completed either because of an error or because the available computer time is inadequate to finish the computation.

The operator or programmer when running a routine on a computer should keep a log of his actions and the performance of the computer. This log should show the times of loading the routine, of starting and ending of computation, and should indicate the settings of all switches and registers on the console at these times. Should an error occur, the time of occurrence and the settings of all switches and registers should be recorded before restarting the computer or before obtaining a listing of

the contents of the memory. The operator's log may be used both in determining causes of errors and in evaluating the performance of the computer.

Routines which will automatically start large computers on one computation after another are used in some computer installations. These routines are called supervisory routines.[1] Problems to be run are arranged sequentially on a magnetic tape along with information on how they are to be initiated. The supervisory routine starts each problem routine. After the problem routine is completed, a transfer is made to the supervisory routine which then starts the processing for the next problem routine. If the computer has a clock and commands for reading the clock, the supervisory routine can also prepare the billings for the computer time used. The human operator is only required to observe any unusual events and to change magnetic tapes and paper.

11-7. Initial Installation of a Computer

After the choice of computer has been made, sufficient time before its delivery should be allowed for preparing floor plans for the computer and auxiliary equipment, for preparing a library of routines and subroutines, and for the training of personnel.

It is desirable to separate the computer from the auxiliary equipment to remove as many distractions as possible from the computer operator. If possible, preparation equipment such as keypunches, verifiers, reproducers, sorters, card-to-paper-tape and paper-tape-to-card equipment, magnetic-tape preparation equipment, and off-line listing equipment should not be placed in the computer room. If there is a lot of this auxiliary equipment, it, too, should be housed in separate rooms according to function. By having a separate room for the computer, the computer operator should normally be the only person in the computer room while a problem is being processed on the computer.

In addition to the rooms for the computer and auxiliary equipment, there should be a storage room for cards, listing paper, paper tapes, and magnetic tapes, a maintenance room adjoining the computer room, and a programmer ready-room where the programmer may assemble his routine before going to the computer room or giving his routine to the computer operator and where he may analyze the results of a computation immediately after the computation results have been listed.

The layout of the computer in the computer room should be such that

[1] In this context *monitor routine* is also frequently used as synonym for *supervisory routine*.

the input and output units and the console are accessible to the computer operator without the operator moving from a central position. That is, he should be able to read the listing of the on-line printer from the console, and he should be able to load an input deck of cards or withdraw an output deck of cards and load or remove a paper or magnetic tape without leaving the area of the console.

The preparing of the library of routines and subroutines and the training of personnel may be undertaken concurrently. The flow diagrams and codes for assembly routines, compiler routines, memory dump routines, trace routines, and the subroutines for the nth root, matrix inversion, roots of polynomials, polynomial fitting of data, and elementary functions, such as exponentials, logarithms, and trigonometric functions, may be used for training class examples as they are developed. In this manner the training may be undertaken with routines the trainees will be using in the near future. Also the programming and coding of the routines are checked thoroughly by the trainees since they must understand each step in the process in order to learn the procedures for programming and coding and to learn the computer.

11-8. Remarks

The reader may find additional references to articles on *Administration and Operation of Computation Centers* and reviews of other computer articles in the new journal *Computing Reviews*, published by the Association for Computing Machinery, and in *Computers and Automation*. Techniques of computer utilization, construction, and theory are changing and improving rapidly. This contributes to making the field of computer sciences one of the most challenging and interesting of our time.

APPENDIX I

THREE-ADDRESS DECIMAL COMPUTER COMMAND LIST

In the description of each order, the three addresses are M_1, consisting of the digits $d_1d_2d_3$; M_2, consisting of the digits $d_4d_5d_6$; and M_3, consisting of the digits $d_7d_8d_9$. The contents of the memory cell addressed M_i is m_i, $i = 1, 2, 3$. The operation is designated by the sign digit s and the digit d_0. The contents of the control counter (abbreviated CC) is the address of the next order to be executed.

A B-boxed order is indicated by replacing the normally even sign digit s by $s + 1$. This instructs the computer that the word following this order is included in the order and indicates which addresses are to be modified by which B-boxes. For example, let L be the address of the B-boxed order

$$\underbrace{sd_0}_{\text{Operation}} \quad \underbrace{d_1d_2d_3}_{M_1} \quad \underbrace{d_4d_5d_6}_{M_2} \quad \underbrace{d_7d_8d_9}_{M_3}$$

where s is an odd digit. Then the following word

$$sd_0 \quad \underbrace{d_1d_2d_3}_{B_{M_1}} \quad \underbrace{d_4d_5d_6}_{B_{M_2}} \quad \underbrace{d_7d_8d_9}_{B_{M_3}}$$

in location $L + 1$, gives the following interpretation to the order addressed L. The address M_1 is modified by adding the digits $b_1b_2b_3$ in the B-box addressed B_{M_1}; the address M_2 is likewise modified by the digits $b_1b_2b_3$ in B-box addressed B_{M_2}; and similarly the address M_3 is modified by the digits $b_1b_2b_3$ in the B-box addressed B_{M_3}. The contents b of a B-box is assumed to be a data word with digits

$$+b_0 \quad \underbrace{b_1b_2b_3}_{\substack{\text{Address} \\ \text{modifier}}} \quad b_4b_5b_6 \quad \underbrace{b_7b_8b_9}_{\substack{\text{Test} \\ \text{address}}}$$

TABLE A-I-1

Symbol	Operation code		Description
	s	d_0	
STP	0	0	Stop operation of computer. If $M_1 = 000$, ignore M_3; if $M_1 = 001$, change CC to M_3. Ignore M_2.
ADD	0	1	$m_3 = m_1 + m_2$; i.e., the result $m_3 = m_1 + m_2$.
SUB	0	2	$m_3 = m_1 - m_2$.
MLR	0	3	$m_3 = m_1 m_2$, where m_3 is the rounded product.
MLT	0	4	High-order digits of $m_1 m_2$ are stored in memory location M_3 and the low-order digits are stored in memory location $M_3 + 1$.
DVR	0	5	$m_3 = m_1/m_2$, where m_3 is the rounded quotient.
DIV	0	6	$m_3 = m_1/m_2$, where m_3 is the unrounded quotient and the remainder times 10^{10} is stored in memory location $M_3 + 1$.
ADA	0	7	$m_3 = \|m_1\| + \|m_2\|$.
SBA	0	8	$m_3 = \|m_1\| - \|m_2\|$.
UCD	2	1	Ignore M_1 and M_2. Next order is taken from memory location M_3.
TRA	2	2	If $m_1 < m_2$, next order is taken sequentially; if $m_1 \geq m_2$, next order is taken from memory location M_3.
OVW	2	3	Ignore M_1 and M_2. If an overflow occurred as the result of the preceding order, take the next order from memory location M_3. If an overflow did not occur, take next order sequentially.
XTR	2	4	Replace the digits of m_3 as designated by m_2 with the corresponding digits of m_1. If in m_2, s or d_r, $r = 0, 1, \ldots, 9$, is a 0, the corresponding sign or digit of m_3 is not changed; and if s or d_r is a 1, the corresponding sign or digit of m_3 is replaced by the corresponding sign or digit of m_1.
SHR	2	5	$m_3 = m_1 \times 10^{-M_2}$.
SHL	2	6	$m_3 = m_1 \times 10^{M_2}$.
SET	2	7	Set B-box with initial value M_1 and final value M_3. The memory cell addressed M_2 is the B-box. This order causes the digits $b_1 b_2 b_3$ of the B-box addressed M_2 to be those of M_1 and the digits $b_7 b_8 b_9$ to be those of M_3, and sets the digits $b_0 = b_4 = b_5 = b_6 = 0$ and $s = +$.
INC	2	8	Increment the B-box addressed M_2 by the amount M_1. This causes $(b_1 b_2 b_3 + M_1) \pmod{1{,}000} \rightarrow b_1 b_2 b_3$. Next, a test is performed to see if $b_1 b_2 b_3 < b_7 b_8 b_9$. If $b_1 b_2 b_3 < b_7 b_8 b_9$, the next order is taken from memory location M_3. If $b_1 b_2 b_3 \geq b_7 b_8 b_9$, the next order is taken sequentially.
DEC	2	9	Decrement the B-box addressed M_2 by the amount M_1. This causes $[b_1 b_2 b_3 + (1{,}000 - M_1)] \pmod{1{,}000} \rightarrow b_1 b_2 b_3$. Next, a test is performed to see if $b_1 b_2 b_3 > b_7 b_8 b_9$. If $b_1 b_2 b_3 > b_7 b_8 b_9$, the next order is taken from memory location M_3. If $b_1 b_2 b_3 \leq b_7 b_8 b_9$, the next order is taken sequentially.

TABLE A-I-1 (*Continued*)

Symbol	Operation code		Description
	s	d_0	
LOD	4	1	Load M_1 words from input unit into memory starting at address M_2 and, when finished, change CC to M_3.
PRT	4	2	Print M_1 words from memory starting at address M_2 and, when finished, change CC to M_3.
SWL	4	3	Connect input unit designated by M_1 to computer and, when finished, change CC to M_3. Ignore M_2.
SWP	4	4	Connect output unit designated by M_1 to computer and, when finished, change CC to M_3. Ignore M_2.
BKP	4	5	If the break-point switch indicated by M_1 is in the ON position, the next order is taken from memory location M_3; i.e., change CC to M_3. If this switch is in the OFF position, the next order is taken sequentially. Ignore M_2.
REW	6	1	Rewind tape on tape unit addressed M_1 and change CC to M_3. Ignore M_2.
FAD	8	1	$m_3 = m_1 + m_2$ where m_3 is the rounded sum and m_1, m_2, and m_3 are in floating point notation.
FSB	8	2	$m_3 = m_1 - m_2$ where m_3 is the rounded difference and m_1, m_2, and m_3 are in floating point notation.
FMR	8	3	$m_3 = m_1 m_2$ where m_3 is the rounded product and m_1, m_2, and m_3 are in floating point notation.
FDR	8	5	$m_3 = m_1/m_2$ where m_3 is the rounded quotient and m_1, m_2, and m_3 are in floating point notation.
FAA	8	7	$m_3 = \|m_1\| + \|m_2\|$ where m_3 is the rounded sum and m_1, m_2, and m_3 are in floating point notation.
FSA	8	8	$m_3 = \|m_1\| - \|m_2\|$ where m_3 is the rounded difference and m_1, m_2, and m_3 are in floating point notation.

Table A-I-1 is in sequence by numeric operation code. For finding the definitions of orders Table A-I-2 supplies the necessary alphabetic cross reference:

TABLE A-I-2

Order		Order		Order	
Symbol	Code	Symbol	Code	Symbol	Code
ADA	07	FSA	88	SET	27
ADD	01	FSB	82	SHL	26
BKP	45	INC	28	SHR	25
DEC	29	LOD	41	STP	00
DIV	06	MLR	03	SUB	02
DVR	05	MLT	04	SWL	43
FAA	87	OVW	23	SWP	44
FAD	81	PRT	42	TRA	22
FDR	85	REW	61	UCD	21
FMR	83	SBA	08	XTR	24

APPENDIX II

TWO-ADDRESS DECIMAL COMPUTER COMMAND LIST

In the description of each order, the two addresses are M_1, consisting of the digits $d_2d_3d_4d_5$, and M_2, consisting of the digits $d_6d_7d_8d_9$. The operation is designated by the sign digit s and the digits d_0d_1. The contents of the memory cell addressed M_1 is m_1. The contents of the upper accumulator A_U is a_U and the contents of the lower accumulator A_L is a_L. The contents of the total accumulator A is a. The contents of the quotient register Q is q. The contents of memory location 0000 is always zero, and this location can be read from, but not read into. The rounded floating point sum, difference, and product are in A_U. The commands FCA and FCS are interchangeable with CAU and CSU, respectively.

In those orders for which M_2 is not the address of the next order, or if $M_2 = 0$, then the next order is taken sequentially.

A B-boxed order is indicated by negating the normally positive sign digit. This instructs the computer that the word following this order is included in the order and indicates which addresses are to be modified by which B-boxes. For example, let L be the address of the B-boxed order

$$\underbrace{sd_0d_1}_{\text{Operation}} \quad \underbrace{d_2d_3d_4d_5}_{M_1} \quad \underbrace{d_6d_7d_8d_9}_{M_2}$$

where $s = -$, then the following word

$$sd_0d_1 \quad \underbrace{d_2d_3d_4d_5}_{B_{M_1}} \quad \underbrace{d_6d_7d_8d_9}_{B_{M_2}}$$

in location $L + 1$, gives the following interpretation to the order addressed L. The address M_1 is modified by adding the digits $b_2b_3b_4b_5$ in the B-box addressed B_{M_1}, and the address M_2 is likewise modified by the digits $b_2b_3b_4b_5$ in B-box addressed B_{M_2}. The contents b of a B-box is assumed to be a data word with digits

$$+b_0b_1 \quad \underbrace{b_2b_3b_4b_5}_{\substack{\text{Address} \\ \text{modifier}}} \quad \underbrace{b_6b_7b_8b_9}_{\substack{\text{Test} \\ \text{address}}}$$

TABLE A-II-1

Symbol	Operation code			Description
	s	d_0	d_1	
STP	+	0	0	Stop computing. List M_1 and m_1 on console typewriter; when computing is started again, next order is taken from memory location M_2 if $M_2 \neq 0$.
CAU	+	0	1	$m_1 \rightarrow a_U$ and $0 \rightarrow a_L$; next order taken from M_2 if $M_2 \neq 0$.
HAU	+	0	2	$a + m_1 \rightarrow a$; next order taken from M_2 if $M_2 \neq 0$.
CMU	+	0	3	$\lvert m_1 \rvert \rightarrow a_U$ and $0 \rightarrow a_L$; next order taken from M_2 if $M_2 \neq 0$.
HMU	+	0	4	$a + \lvert m_1 \rvert \rightarrow a$; next order taken from M_2 if $M_2 \neq 0$.
CAL	+	0	5	$m_1 \rightarrow a_L$ and $0 \rightarrow a_U$; next order taken from M_2 if $M_2 \neq 0$.
HAL	+	0	6	$a + 10^{-10} m_1 \rightarrow a$; next order taken from M_2 if $M_2 \neq 0$.
CML	+	0	7	$\lvert m_1 \rvert \rightarrow a_L$ and $0 \rightarrow a_U$; next order taken from M_2 if $M_2 \neq 0$.
HML	+	0	8	$a + 10^{-10} \lvert m_1 \rvert \rightarrow a$; next order taken from M_2 if $M_2 \neq 0$.
CSU	+	1	1	$-m_1 \rightarrow a_U$ and $0 \rightarrow a_L$; next order taken from M_2 if $M_2 \neq 0$.
HSU	+	1	2	$a - m_1 \rightarrow a$; next order taken from M_2 if $M_2 \neq 0$.
CSL	+	1	3	$-m_1 \rightarrow a_L$ and $0 \rightarrow a_U$; next order taken from M_2 if $M_2 \neq 0$.
HSL	+	1	4	$a - 10^{-10} m_1 \rightarrow a$; next order taken from M_2 if $M_2 \neq 0$.
CAQ	+	2	1	$m_1 \rightarrow q$; next order taken from M_2 if $M_2 \neq 0$.
CMQ	+	2	2	$\lvert m_1 \rvert \rightarrow q$; next order taken from M_2 if $M_2 \neq 0$.
CSQ	+	2	3	$-m_1 \rightarrow q$; next order taken from M_2 if $M_2 \neq 0$.
MLR	+	3	1	$r + qm_1 \rightarrow a$, $r = \pm 00\ 0000\ 0000\ 50\ 0000\ 0000$ where $r > 0$ if $qm_1 \geq 0$ and $r < 0$ if $qm_1 < 0$; next order taken from M_2 if $M_2 \neq 0$.
MLT	+	3	2	$qm_1 \rightarrow a$; next order taken from M_2 if $M_2 \neq 0$.
DVR	+	3	3	$a/m_1 \rightarrow q$, where q is the rounded quotient; next order taken from M_2 if $M_2 \neq 0$.
DIV	+	3	4	$a/m_1 \rightarrow q$, where q is the unrounded quotient and, at end of operation, a_U contains 10^{10} times the remainder and $a_L = 0$; next order taken from M_2 if $M_2 \neq 0$.
SHR	+	3	5	$10^{-M_1} a \rightarrow a$; next order taken from M_2 if $M_2 \neq 0$. Digits shifted out of A are lost.
SHL	+	3	6	$10^{M_1} a \rightarrow a$; next order taken from M_2 if $M_2 \neq 0$. Digits shifted out of A are lost.
STU	+	4	1	$a_U \rightarrow m_1$; next order taken from M_2 if $M_2 \neq 0$.
STL	+	4	2	$a_L \rightarrow m_1$; next order taken from M_2 if $M_2 \neq 0$.
STQ	+	4	3	$q \rightarrow m_1$; next order taken from M_2 if $M_2 \neq 0$.
TAP	+	5	1	If $a > 0$, next order taken from M_1; if $a \leq 0$, next order taken from M_2 if $M_2 \neq 0$.

TABLE A-II-1 (*Continued*)

Symbol	Operation code			Description
	s	d_0	d_1	
TAN	+	5	2	If $a < 0$, next order taken from M_1; if $a \geq 0$, next order taken from M_2 if $M_2 \neq 0$.
TAZ	+	5	3	If $a = 0$, next order taken from M_1; if $a \neq 0$, next order taken from M_2 if $M_2 \neq 0$.
OVW	+	5	4	If an overflow occurred in the result of the previous command, take next order from M_1. If an overflow did not occur, take next order from M_2 if $M_2 \neq 0$.
XTR	+	5	5	Replace the sign and digits of a_U as designated by q with the sign and corresponding digits of m_1. If in q, $s = +$ or $q_i = 0$, the sign or corresponding digit of a_U is left unchanged; if in q, $s = -$ or $q_i = 1$, the sign or corresponding digit of a_U is replaced by the sign or corresponding digit of m_1. The next order is taken from M_2 if $M_2 \neq 0$.
SBI	+	5	6	Set B-box addressed M_1 with initial value M_2. This order causes the digits, $b_2b_3b_4b_5$, of the B-box to be those of M_2, $s = +$, $b_0 = b_1 = 0$, and leaves the remaining digits unchanged. The next order is taken sequentially.
SBF	+	5	7	Set B-box addressed M_1 with final value M_2. This order causes the digits, $b_6b_7b_8b_9$, of the B-box to be those of M_2 and leaves the remaining digits unchanged. The next order is taken sequentially.
INC	+	5	8	Increment the B-box addressed M_1 by the amount M_2. This causes $[b_2b_3b_4b_5 + M_2] (\text{mod } 10{,}000) \rightarrow b_2b_3b_4b_5$. Next, a test is performed to see if $b_2b_3b_4b_5 < b_6b_7b_8b_9$. If $b_2b_3b_4b_5 < b_6b_7b_8b_9$, the next order is taken sequentially. If $b_2b_3b_4b_5 \geq b_6b_7b_8b_9$, the next order is taken from the memory location next after the sequential location, i.e., skip the next sequential command.
DEC	+	5	9	Decrement the B-box addressed M_1 by the amount M_2. This causes $[b_2b_3b_4b_5 + (10{,}000 - M_1)] (\text{mod } 10{,}000) \rightarrow b_2b_3b_4b_5$. Next a test is performed to see if $b_2b_3b_4b_5 > b_6b_7b_8b_9$. If $b_2b_3b_4b_5 > b_6b_7b_8b_9$, the next order is taken sequentially. If $b_2b_3b_4b_5 \leq b_6b_7b_8b_9$, the next order is taken from the memory location next after the sequential location, i.e., skip the next sequential command.
UCD	+	6	1	Take next command from memory location M_2. Ignore M_1.
LOD	+	7	1	Load M_1 words from input unit into memory starting at address M_2, $M_2 \neq 0$. Next order is taken sequentially.

TABLE A-II-1 (*Continued*)

Symbol	Operation code			Description
	s	d_0	d_1	
PRT	+	7	2	Print M_1 words from memory starting at address M_2. Next order is taken sequentially.
SWL	+	7	3	Connect input unit designated by M_1 to computer. Next order taken from M_2 if $M_2 \neq 0$.
SWP	+	7	4	Connect output unit designated by M_1 to computer. Next order taken from M_2 if $M_2 \neq 0$.
BKP	+	7	5	If the break-point switch indicated by M_1 is in the ON position, the next order is taken from memory location M_2. If this switch is in the OFF position, the next order is taken sequentially.
FCA	+	8	1	$m_1 \rightarrow a_U$, where m_1 and a_U are in floating point notation. Next order taken from M_2 if $M_2 \neq 0$.
FHA	+	8	2	$a_U + \lvert m_1 \rvert \rightarrow a_U$, where m_1 and a_U are in floating point notation. Next order taken from M_2 if $M_2 \neq 0$.
FMA	+	8	3	$a_U + \lvert m_1 \rvert \rightarrow a_U$, where m_1 and a_U are in floating point notation. Next order taken from M_2 if $M_2 \neq 0$.
FCS	+	8	4	$-m_1 \rightarrow a_U$, where m_1 and a_U are in floating point notation. Next order taken from M_2 if $M_2 \neq 0$.
FHS	+	8	5	$a_U - m_1 \rightarrow a_U$, where m_1 and a_U are in floating point notation. Next order taken from M_2 if $M_2 \neq 0$.
FMR	+	8	6	$m_1 q \rightarrow a_U$, where a_U is the rounded product and m_1, q, and a_U are in floating point notation. Next order taken from M_2 if $M_2 \neq 0$.
FDR	+	8	7	$a_U / m_1 \rightarrow q$, where q is the rounded quotient and a_U, m_1, and q are in floating point notation. Next order taken from M_2 if $M_2 \neq 0$.
REW	+	9	1	Rewind tape on tape unit addressed M_1. Next order taken from M_2 if $M_2 \neq 0$.

Table A-II-2 provides an alphabetic cross reference for the two-address orders.

TABLE A-II-2

Order		Order		Order	
Symbol	Code	Symbol	Code	Symbol	Code
BKP	+75	FHA	+82	REW	+91
CAL	+05	FHS	+85	SBF	+57
CAQ	+21	FMA	+83	SBI	+56
CAU	+01	FMR	+86	SHL	+36
CML	+07	HAL	+06	SHR	+35
CMQ	+22	HAU	+02	STL	+42
CMU	+03	HML	+08	STP	+00
CSL	+13	HMU	+04	STQ	+43
CSQ	+23	HSL	+14	STU	+41
CSU	+11	HSU	+12	SWL	+73
DEC	+59	INC	+58	SWP	+74
DIV	+34	LOD	+71	TAN	+52
DVR	+33	MLR	+31	TAP	+51
FCA	+81	MLT	+32	TAZ	+53
FCS	+84	OVW	+54	UCD	+61
FDR	+87	PRT	+72	XTR	+55

INDEX